VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KOMPILÁTOR JAZYKA C PRO VLIW ARCHITEKTURY

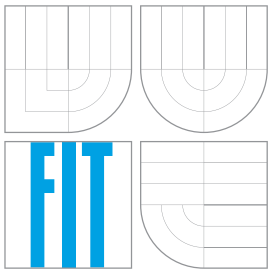
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ MINÁČ

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KOMPILÁTOR JAZYKA C PRO VLIW ARCHITEKTURY

C COMPILER FOR VLIW ARCHITECTURES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ MINÁČ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. KAREL MASAŘÍK, Ph.D.

BRNO 2013

Abstrakt

Tato práce pojednává o jazyce CodAl a Codasip frameworku. Dále popisuje kompilační platformu LLVM, jazyk LLVM IR a optimalizace nad tímto jazykem. Vytvoření návrhu a implementace rozšíření kompilační platformy LLVM o globální plánovaní instrukcí na základe profilu je cílem této práce.

Abstract

This work discusses about CodAl language and Codasip framework. It describes LLVM compiling platform, LLVM IR and its possible optimizations. The result of this work is creation and implementation a proposal of global scheduling dependence on profile as extension in LLVM.

Klíčová slova

Plánování, překladač, VLIW architektura, základní blok, globální plánovaní, LLVM, Codasip.

Keywords

Scheduling, compiler, VLIW architecture, Basic Blok, Global Scheduling, LLVM, Codasip.

Citace

Tomáš Mináč: Kompilátor jazyka C pro VLIW architektury, diplomová práce, Brno, FIT VUT v Brně, 2013

Kompilátor jazyka C pro VLIW architektury

Prohlášení

Prohlašuji, že jsem tuto diplomovú práci vypracoval samostatně pod vedením pana Karle Masaříka.

.....
Tomáš Mináč
20. května 2013

Poděkování

Chtěl bych poděkovat celému týmu Lissom za umožnění spolupráce na jejich projektu. Jmenovitě děkuji panu doktoru Masaříkovi za rady a pomoc při psaní práce. Rodičům děkuji za podporu během celého studia.

© Tomáš Mináč, 2013.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Dnešné procesory sú optimalizované pre vysoký výkon a nízky príkon. V oblasti univerzálnych procesorov prevládajú viac–jadrové superskalárne procesory. Architektúra VLIW nie je zrovna vlajkovou loďou veľkých výrobcov procesorov.

Je zaujímavé, že aj superskalárne a VLIW architektúry sa snažia využiť potenciálu paralelizmu na úrovni inštrukcií (ILP). Superskalárna architektúra úlohu nájdenia paralelizmu rieši na hardvérovej úrovni, zatiaľ čo architektúra VLIW prenecháva túto úlohu kompilátoru. Tým by sa mal zjednodušiť hardvér a zároveň aj znížiť príkon. Je ale známe, že nie každá aplikácia disponuje dostatočným ILP. VLIW architektúry sa osvedčujú v oblasti spracovania videa alebo obrazových dát.

Myšlienka využitia tejto práce je vytvoriť jednoduchý VLIW procesor v Codasip frameworku určený pre spracovanie videa napríklad pomocou štandardu *MPEG–4*. Pre tento procesor by mal byť prekladač generovaný a zároveň schopný na základe profilu optimalizovať prekladaný program. Zároveň by malo byť možné optimalizovať na základe výsledkov prekladu vytváranú architektúru.

Táto práca z časti nadväzuje a prepracováva moju bakalársku prácu, ktorá pojednáva tiež o architektúre VLIW a tvorbe prekladača, no pokračuje tam kde predchádzajúca práca skončila. Dokončuje myšlienku a navrhuje implementáciu globálneho plánovania. Práca však nepodáva zoznam známych VLIW architektúr ani ich vlastností, lebo tento zoznam a popis obsahuje spomínaná bakalárska práca [15].

Práca tiež pojednáva o frameworku Codasip, v ktorom je možné vytvárať procesory s podporou potrebných nástrojov, ďalej o kompilačnom frameworku LLVM, v ktorom je prekladač realizovaný.

Posledná časť sa venuje návrhu a vypracovaniu zadania. Detailne popisuje navrhnuté a implementované algoritmy. Práca využíva a spája to čo už LLVM obsahuje s navrhnutými rozšíreniami a novým programom pre tvorbu *superblokov*. Tiež sa venuje popisu známych obmedzení a problémov navrhnutého riešenia. Nakoniec testovaniu s vyhodnotením výsledkov.

Kapitola 2

Nástroje a architektúra použité pre vývoj

2.1 CodAl

Jazyk **CodAl** je vyvíjaný za účelom rýchleho prototypovania viacprocesorových systémov na čipe (MPSoC) alebo aplikačne špecifických inštrukčne definovaných procesorov (ASIP). Vychádza z jazyku **LISA** (**L**anguage for **I**nstruction–**S**et **A**rchitecture) a spadá do kategórie jazykov popisujúcich architektúru. Z popísaného modelu daného procesoru je možné automaticky generovať nástroje pre programovanie, simuláciu a popis implementácie mikro–architektúry v jazyku VHDL.

Model popísaný jazykom CodAl sa preloží pomocou Codasip prekladača. Prekladač kontroluje syntax a je schopný detekovať niektoré chyby v popise modelu. Napríklad, ak inštrukcia nemá jedinečný binárny formát. Výstupom je popis pomocou XML, ktorý je vstupom pre generátor nástrojov.

Na to, aby bolo možné model vôbec preložiť je nutné, aby obsahoval dve nasledovné časti.

- Popis zdrojov (Resources description) – obsahuje popis hardvérových prvkov, ktoré má procesor k dispozícií. To znamená definovať registre, pamäte, zbernice ... Je povinné definovať *programový čítač* (program counter), *pamäťový zdroj* (memory resource) a *mapovanie adresového priestoru do pamäte* (memory mapping).

- Inštrukcie a popis udalostí (Operations section) – obsahuje popis inštrukčnej sady a popis reakcie pre udalosti, ako napríklad načítanie alebo dekódovanie inštrukcie (instruction decode, instruction fetching). Udalosti, ktorých popis musí byť obsiahnutý v modeli sú *reset*, *halt*, *main* a popis samotných inštrukcií procesoru.

Pre lepšiu predstavu ako vyzerá popis modelu v jazyku CodAl uvádzam na obrázku 2.1 jednoduchý príklad prevzatý z [4]. Na začiatku je nutné definovať programový čítač *pc* v našom prípade 8 bitový. Pamäť *ram* obsahuje 256 blokov, kde každý blok obsahuje 8 bitov. Nad pamäťou je možné vykonávať operácie typu *read, write* a *execute*. *lau* znamená *least addressable unit* teda najmenšia možná adresovateľná jednotka je 8 bitov a celá pamäť je organizovaná ako *little–endian*. Nasleduje popis zbernice *bus*. Zbernica musí byť organizovaná tiež ako *little–endian*, lebo je pripojená k pamäti. Posledná časť popisu zdrojov definuje mapovanie adries pamäte. Nakoľko je pamäť prepojená s jadrom pomocou zbernice, je nutné jej prideliť adresy. V našom prípade sú to adresy v rozmedzí 0 až 255.

```
program_counter bit[8] pc;              //the program counter
memory bit[8] ram {                     //memory resource
      .endianess = little,
      .lau = 8,                         //size of a block
      .size = 256,                      //number of blocks
      .flags = {r,w,x}
};

bus bit[8] bus {                        //bus resource
      .lau = 8,
      .endianess = little
};
memorymapping defaultmap {              //memory mapping
      bus bus: 0..255 = memory[7..0];   //memory 8b wide->256 addresses
}
element inop {                          //the element
      assembler {"nop"};                //textual representation
      binary {0b0000};                  //binary element code

}                                       //element inop
event halt {}                           //stop simulation
element ihalt {
      use halt;                   //element halt will be used locally
      assembler {"halt"};
      binary {0b1111};
      timing {halt;};                   //halt should be executed
}                                       //element ihalt

set instructions = inop, ihalt;

event main {
      use instructions;
      start {{instructions;}};
      decoders (pc) {{instructions(bus[pc]);}};
} //event main

event reset {
      semantics {
            pc = 0;               //reset program counter
      } //behavior section
} //event reset
```

Obrázok 2.1: Príklad jednoduchého modelu v jazyku CodAl. Príklad je prevzatý z [4].

Ďalej nasleduje popis inštrukcií (elementov) a udalostí. Ako prvá sa popisuje inštrukcia *inop*, ktorá má definovaný binárny zápis a asemblerovskú syntax. Ďalej je definovaná udalosť *halt*, ktorá je vložená do inštrukcie *ihalt*. Nakoľko udalosť *halt* je povinná v každom modeli, má špeciálnu sémantiku, ktorá modeluje zastavenie procesoru. V inštrukcií *ihalt* okrem asemblerovskej syntaxy a binárneho zápisu definujeme aj udalosť, ktorá sa má vykonať (časť *timing*).

Dva definované elementy zlúčime do množiny elementov *instructions*. Nasleduje popis udalosti *main*. Táto udalosť je tiež povinná a je spustená pri každom časovom cykle. Jej hlavnou úlohou je spustenie vykonávania ďalšej inštrukcie. Samotné dekódovanie inštrukcie je popísane v sekcií *start* a *decoders*. Poslednou udalosťou je *reset*, má definovanú len sémantiku teda akciu, ktorá sa má vykonať v prípade spustenia udalosti. V našom príklade sa programový čítač nastaví na nulu čo znamená, že prevádzaný program sa začne vykonávať od začiatku.

Pre priehľadnejší model a popis je možné uložiť popis sémantiky do **.cpp* a **.h* súborov, ktoré sú potom do modelu prilinkované. Pre popis sémantiky v modeli a aj dodatočných súborov sa používa podmnožina jazyka *ANSI C*. Nie sú podporované pointre, štruktúry, príkaz *goto*, nie je možné deklarovať a definovať premennú na tom istom riadku popisu (int x=0;), ...

V jazyku CodAl je možné popísať jednu architektúru na dvoch úrovniach. Na úrovni inštrukčnej sady (instruction accurate model) alebo na úrovni jednotlivých cyklov (cycle accurate model). Model popísaný na úrovni inštrukčnej sady abstrahuje nad detailami architektúry zatiaľ čo model na úrovni cyklov popisuje každú potrebnú komponentu detailne. Z jednotlivých úrovní popisu sa generujú rôzne nástroje, čo je predmetom textu nasledujúcej sekcie.

2.2 Codasip framework

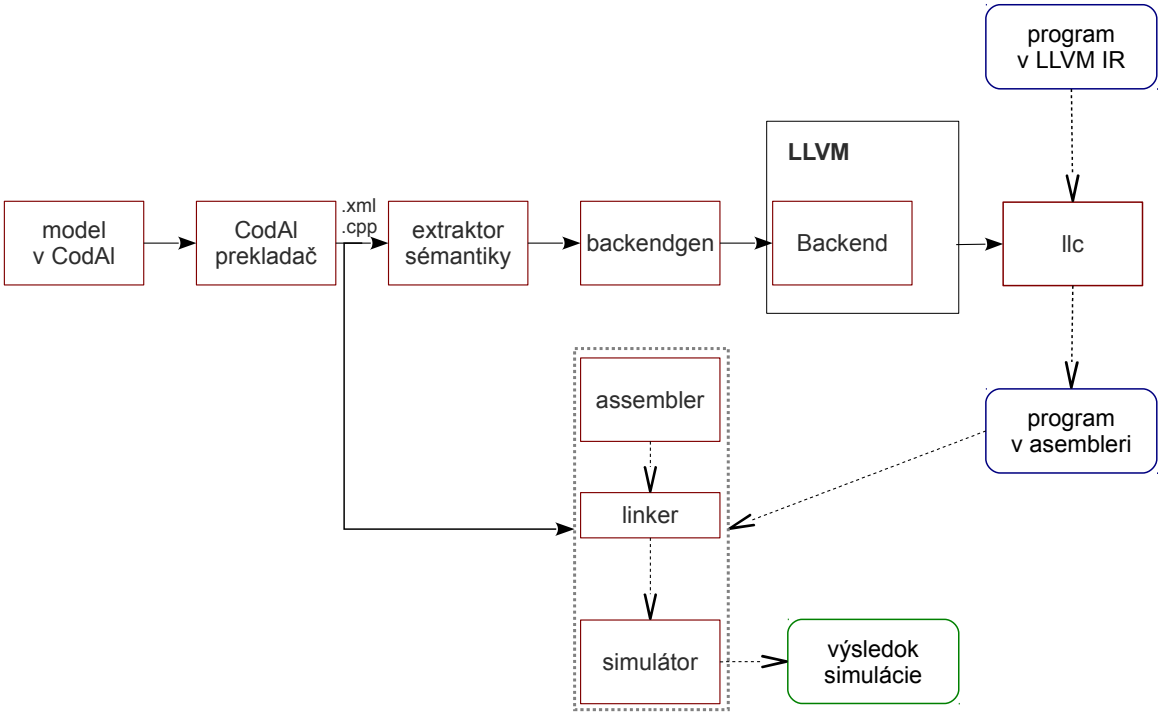
Codasip framework [5] poskytuje nástroje pre prácu z vyššie popísaným jazykom CodAl. Poskytuje samotný prekladač z jazyka CodAl do XML reprezentácie, ktorá je vstupom pre generátor nástrojov. Dostupné sú nasledovné nástroje:

- assembler – nástroj, ktorý slúži pre preklad programu z jazyka asembler danej architektúry do binárneho objektového súboru, ktorý je potom linkovaný pomocou nástroja Codasip Linker do binárnej formy určenej priamo pre procesor,
- linker – spája niekoľko binárnych objektových súborov do jedného binárneho súboru určeného priamo pre daný procesor,
- prekladač z jazyka C – slúži pre preklad z jazyka C do jazyka asembler danej architektúry. Jeho generovanie pozostáva z dvoch krokov. Najprv je nutné vyextrahovať sémantiku inštrukcií z modelu procesoru. Z vyextrahovanej sémantiky je možné generovať priamo kompilátor, tento proces bude ešte bližšie popísaný,
- simulátor – je možné vygenerovať hneď niekoľko druhov simulátorov a to interpretovaný, kompilovaný, simulátor na RTL úrovni,
- profiler – sleduje beh simulácie a zaznamenáva dôležité informácie pre ladenie programu,
- debugger – užitočný nástroj pri ladení programu, ktorý umožňuje vkladanie tzv. break–pointov, zmenu hodnôt v zdrojoch procesoru (registrov, pamäte), čítanie týchto hodnôt a vyhodnocovanie výrazov nad týmito hodnotami,
- disassembler – nástroj, ktorý umožňuje spätný preklad z binárnej reprezentácie programu späť do reprezentácie v asembleru,
- generovanie popisu architektúry v HDL jazyku.

Nedeliteľnou súčasťou Codasip frameworku je Codasip-studio. Grafické užívateľské prostredie založené na open–source vývojovej platforme Eclipse. Dopĺňa svoje vlastné pluginy a rozšírenia, čím umožňuje editáciu a celkovú prácu s modelmi, dodatočnými súbormi, generovanie a použitie nástrojov.

2.2.1 Preklad a simulácia programov

Nakoľko sa moja práca venuje hlavne prekladu programov, bližšie popíšem postup generovania prekladača a samotný preklad generovaným prekladačom. Budem postupovať podľa obrázku 2.2 cez plné šípky. Máme model nášho procesoru v jazyku CodAl a niekoľko dodatočných súborov. Pomocou CodAl prekladaču dostaneme popis v *.xml*. Práve tento popis a dodatočné súbory k modelu sú vstupom pre extraktor sémantiky.



Obrázok 2.2: Vytvorenie backendu. Postup prekladu a simulácie pomocou Codasip framework.

Extraktor sémantiky je samostatná časť v Codasip framework. Je ho možné použiť len nad modelom popísanom na *instruction accurate* úrovni. Je založený na LLVM (sekcia 2.3). Skladá sa z niekoľkých priechodov. Prvé priechody spracovávajú vstupný model a rozlišujú jednotlivé inštrukcie. Napríklad ak inštrukcia *add* má niekoľko foriem, teda je ju možné použiť nad dvomi registrami, alebo registrom a operandom vyjadreným číslom v extrahovanej sémantike bude obsiahnutá dvakrát. Takýmto spôsobom sa zaznamená každá varianta danej inštrukcie. Samotná sémantika je popísaná v jazyku *ANSI C*. Z jazyku C sa preloží do jazyku *LLVM IR* kap. 2.3.1, ktorému sa budem venovať ešte neskôr. Nad zápisom sémantiky v *LLVM IR* sa prevádzajú postupne rôzne optimalizácie, aby získaná sémantika mala čo najjednoduchší charakter. Nie však všetky informácie je možné držať v tomto zápise. Z *LLVM IR* sa posledným priechodom sémantika zapíše do špeciálneho tvaru, ktorý pre jednotlivé inštrukcie obsahuje samotnú sémantiku inštrukcie a ďalšie informácie ako napríklad syntax v jazyku asembler daného procesoru, binárne kódovanie, latenciu, ak sa jedná o VLIW architektúru tak možnú pozíciu v bundle a dodatočné poznámky o inštrukcií.

Súbor s extrahovanou sémantikou je vstupný súbor pre generátor backendu (ďalej už len backendgen). Backendgen je program, ktorý načíta extrahovanú sémantiku a na základe

nej vygeneruje backend podľa platformy LLVM, čiže vygeneruje zdrojové súbory v jazyku C++. Tieto súbory sa spolu s knižnicami z LLVM preložia do samostatného programu — výsledného prekladača.

Samotný preklad aplikácie pomocou generovaného prekladača vyzerá nasledovne, budem sledovať prerušované šípky v obrázku 2.2. Najprv je nutné program v jazyku C/C++ preložiť pomocou frontendu do medzikódu *LLVM IR* kap. 2.3.1 (tento proces nie je na obrázku naznačený). Ako frontend môžme použiť prekladač *clang* [2] alebo upravené *llvm–gcc*. Ako je náhle program v *LLVM IR* môžme ho preložiť pomocou vygenerovaného prekladača do jazyku symbolických adries daného procesoru. Tento program pomocou nástroja *assembler* preložíme do binárneho objektového formátu a pomocou nástroju *linker* do binárneho formátu určeného priamo pre daný procesor. Preložený program je možné odsimulovať použitím nástroja *simulátor*, z ktorého získame výsledky simulácie, teda výpis všetkých registrov, návratovú hodnotu programu, informáciu či simulácia dopadla v poriadku alebo skončila chybou, dĺžku simulácie v sekundách, počet procesorových cyklov a konečne rýchlosť simulácie v MHz.

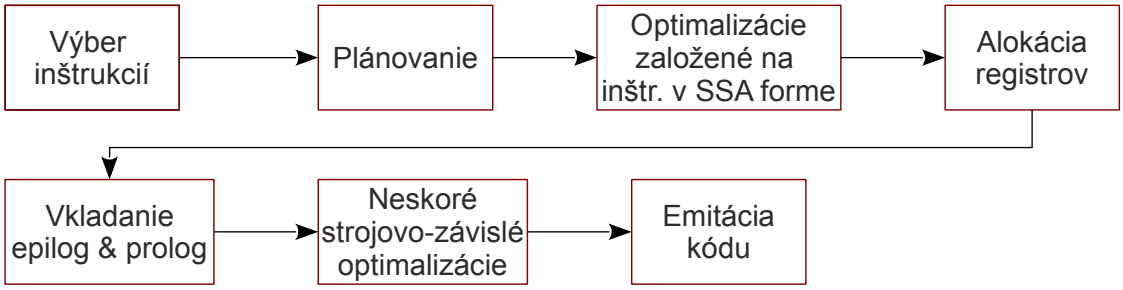
2.3 LLVM

Low level virtual machine je kompilačná platforma obsahujúca kolekciu knižníc a nástrojov, ktoré uľahčujú stavbu prekladačov, optimalizátorov, JIT kód generátorov a programov súvisiacich s prekladačmi. Medzi silné stránky LLVM infraštruktúry patrí okrem modularity, jednoduchého dizajnu, aj nezávislosť na koncovej platforme, zameranie na optimalizácie a iné [13].

Ak si pripomenieme základnú schému prekladača obr. 2.3, tak vstupný program je preložený najprv pomocou frontendu do medzikódu. V našom prípade je možné použiť program *clang* [2] alebo *llvm–gcc*. Výstup prekladu je program v medzikóde v našom prípade *LLVM IR* kap. 2.3.1. Nad programom v medzikóde je možné previesť niekoľko optimalizácií napríklad propagáciu konštánt, odstraňovanie mŕtveho kódu a ďalšie – bližšie popísané v kapitole 2.3.2. V LLVM na aplikovanie optimalizácií slúži program *opt*, ktorému cez parameter je možné nastaviť postupnosť optimalizačných priechodov.

Obrázok 2.3: Základná schéma prekladača.

Vstupom pre poslednú časť prekladača (backend) je optimalizovaný medzikód. Úlohou backendu je transformovať medzikód do asembleru danej architektúry alebo priamo do binárnej formy. V LLVM je backend reprezentovaný programom *llc*. Samotná transformácia je rozdelená do niekoľkých častí (ilustrované na 2.4). Najprv sa nad programom v medzikóde prevedie výber inštrukcií. Program je rozdelený na základné bloky (basic blocks), jednotlivé inštrukcie v medzikóde sa transformujú na inštrukcie danej architektúry. Je časté, že niekoľko inštrukcií v medzikóde sa transformuje na jedinú inštrukciu koncovej architektúry.



Obrázok 2.4: Postupnosť priechodov v backende LLVM.

Tiež sa môže stať opačný prípad, kedy jedna inštrukcia medzikódu sa transformuje na viaceré inštrukcie koncovej architektúry. Obecne sa jedná o úplný NP problém.

Ďalej sa nad inštrukciami, ktoré už majú formu asembleru koncovej architektúry, ale ešte stále obsahujú virtuálne registre, prevedie plánovanie. Dôležitý je fakt, že v LLVM sa plánovanie prevádza nad základnými blokmi a je možné vybrať niekoľko možností, kde väčšina je založená na algoritme *List Scheduling*. Ďalší dôležitý fakt je, že inštrukcie obsahujú virtuálne registre, pomocou ktorých je zabezpečená dátová závislosť jednotlivých inštrukcií a zároveň plánovanie nie je omedzené počtom registrov.

Po plánovaní je posledná možnosť previesť posledné optimalizácie založené na tom, že inštrukcie sú v SSA[1] forme.

Ďalej nasleduje alokácia registrov, kedy každému virtuálnemu registru sa priradí fyzický register danej architektúry. Ak by bol počet fyzických registrov nedostatočný je nutné danú hodnotu uložiť do pamäte, čo vo výsledku často býva oveľa drahšia operácia. Problém optimálneho priradenia virtuálnych registrov fyzickým je opäť NP úplný.

Vkladanie prologu a epilógu vkladá inštrukcie pred volaním a po volaní funkcií (sub–rutín). Každá architektúra má k volaniu funkcií rôzny prístup. Principiálne je nutné predať parametre funkcií a prevziať výsledok, ďalej ošetriť prepísanie živých registrov volajúcej funkcie volanou. Pojmom živý register sa myslí register, ktorý uchováva hodnotu, ktorá sa vo volajúcej funkcií po zavolaní volanej funkcií ešte použije.

Časť neskorých strojovo–závislých optimalizácií umožňuje volať optimalizačné priechody, ktoré sú špecifické pre danú architektúru. Napríklad chytré vkladanie prázdnych inštrukcií na miesto hazardov medzi základnými blokmi a iné.

Emitácia kódu znamená tisk asemblerovských inštrukcií programu pre danú architektúru alebo priamo binárnu podobu programu.

2.3.1 LLVM IR

Informácie v tejto časti sú čerpané z [1]. *LLVM IR* je troj–adresný kód založený na SSA forme, v LLVM manuáloch označovaný ako LLVM asembler. Je tu snaha udržať túto reprezentáciu programu pri nízkej abstrakcií a zároveň musí byť možné preložiť do *LLVM IR* program zapísaný vo vyššom jazyku ako napr. C++. *LLVM IR* je navrhnutý tak, aby ho bolo možné použiť v troch rôznych formách:

- preklad programu do asembleru danej architektúry alebo binárnej formy,

[1] Static single assigment form znamená, že do jednej premennej je možné priradiť v celom programe hodnotu len jedenkrát čo zvyšuje možnosti optimalizácie.

- bitkódová reprezentácia určená pre spracovanie virtuálnym strojom,
- kód musí byť zobraziteľný v čitatelnej forme pre človeka.

Kód je formovaný do *modulov*. Nad modulmi pracuje LLVM linker, ktorý je schopný moduly spájať a vytvárať výsledný súbor prekladaného programu. Modul sa skladá z funkcií, ktoré reprezentujú funkcie zapísané v pôvodnom vyššom jazyku. To znamená, že ak sme mali v pôvodnom programe funkciu s názvom `addTwoNumbers` v medzikóde nájdeme funkciu s rovnakým názvom.

Funkcie sa ďalej skladajú zo základných blokov (basic blocks). Základný blok je taký úsek kódu, do ktorého je možné vojsť len prvou inštrukciou a opustiť ho len poslednou inštrukciou. Teda do tohoto úseku kódu existuje len jeden vstup a neexistujú žiadne postranné výstupy. Jednotlivé základné bloky sa skladajú zo samotných inštrukcií *LLVM IR*.

Pre lepšiu predstavu uvediem malý príklad na obrázku 2.5. Kód je súčasťou cyklu a stará sa o inkrementáciu premennej cyklu. *for.inc:* je názov návestia a zároveň názov základného bloku. Prvou inštrukciou je inštrukcia *load*, ktorá načíta z pamäte 32 bitovú hodnotu typy integer. Môžme si všimnúť, že premenné majú pred sebou znak *%*. To znamená, že sa jedná o lokálne premenné funkcie, kde sa kód nachádza. Globálne premenné začínajú znakom *@*. Takže lokálny pointer je reprezentovaný *%j*. Ďalej je uvedené zarovnanie *align* premennej v pamäti.

```
for.inc:                                          ; preds = %for.body3
  %5 = load i32* %j, align 4
  %inc = add i32 %5, 1
  store i32 %inc, i32* %j, align 4
  br label %for.cond1
```

Obrázok 2.5: Ukážka kódu v *LLVM IR*.

Ďalšia inštrukcia pričíta k načítanej hodnote z predchádzajúcej inštrukcie jednotku. Výsledok bude 32 bitový typu integer. Inštrukcia *store* výsledok sčítania uloží opäť naspäť do pamäte na miesto, kde ukazuje pointer *%j*. Posledná inštrukcia základného bloku je skoková inštrukcia. V rámci *LLVM IR* má inštrukcia *br* dva možné tvary. Jeden je podmienený a skok závisí na podmienke. V príklade je použitý tvar, ktorý reprezentuje jednoduchý nepodmienený skok na návestie *%for.cond1*.

LLVM IR obsahuje niekoľko desiatok inštrukcií, ktorých celkový zoznam sa nachádza v [1]. Pre ďalší text je však dôležité objasniť inštrukciu *phi*. Táto inštrukcia implementuje φ uzol v SSA grafovej reprezentácií funkcie. Jej syntax je nasledovná:
`<result> = phi <ty> [ <val0>, <label0>], ...`

- `<result>` – názov premennej do ktorej sa uloží výsledok,
- `=` – priradenie,
- `phi` – názov inštrukcie,
- `<ty>` – typ výslednej premennej,
- `<val0>` – hodnota, ktorá sa uloží do premennej,

- `<label0>` – návestie, resp. názov základného bloku vo funkcií,
- `...` – inštrukcia môže obsahovať viacero hranatých zátvoriek so svojim obsahom.

Typicky sa *phi* inštrukcia nachádza na začiatku základného bloku a do výslednej premennej ukladá hodnotu z dvojice z hranatých zátvoriek. Vyberá tú dvojicu, ktorej názov základného bloku odpovedá názvu základného bloku z ktorého vykonávanie programu prešlo do aktuálneho základného bloku. Teda niekoľko základných blokov končí skokovou inštrukciou *br*, ktorej cieľ je základný blok s prvou *phi* inštrukciou s názvom napr. *exitBlock*. Výsledná hodnota sa určí podľa toho, z ktorého základného bloku prešlo vykonávanie programu do *exitBlocku*.

2.3.2 Optimalizácie nad LLVM IR

Optimalizácie nad LLVM asemblerom sú implementované ako priechody. Načítajú vstupný súbor v *LLVM IR* prevedú sa rôzne transformácie alebo analýzy a výstupom je zmenený súbor v *LLVM IR*. Dostupné optimalizácie v LLVM je možné rozdeliť do troch skupín [17].

Prvá skupina sú analyzačné priechody. Tieto priechody prechádzajú program a zbierajú informácie, ktoré je možno potom použiť v ďalších priechodov. Sami o sebe nemenia vstupný kód.

Druhou skupinou sú priechody ktoré menia kód tzv. transformačné priechody. Často pred samotným použitím najprv púšťajú analyzačné priechody.

Poslednou skupinou sú tzv. utility priechody. Ich činnosť môže byť rôzna, ale v konečnom dôsledku ich spustením dosiahneme zmenu kódu, ale nespadajú do prvých dvoch skupín. Napríklad priechod, ktorý uloží program do bitkódu alebo ho z neho načíta.

Nasleduje krátky zoznam priechodov s popisom, ktoré som pri návrhu a implementácií použil:

- mem2reg – vyhľadá *alloca* inštrukcie, ktorých výsledok sa používa len inštrukciami *load* a *store* tieto odstráni a namiesto nich sa uloží hodnota do registru, teda namiesto použitia pamäte pre uloženie danej hodnoty sa použije register,
- reg2mem – je reverzná operácia ku predchádzajúcej *mem2reg*,
- loop-siplify – snaží sa previesť prirodzené cykly do jednotnej formy čo zjednodušuje prácu ďalším optimalizáciam,
- loop-rotate – jedná sa o rotáciu cyklov, čiže cyklus, ktorý ma podmienku na začiatku sa môže transformovať na cyklus s podmienkou na konci,
- lcssa – pridáva *phi* inštrukcie na koniec cyklu pre všetky premenné, ktorých hodnota je ešte potrebná aj za hranicami cyklu, funkčnosť sa tým nezmení, transformácia má zmysel pre ďalšie optimalizačné priechody,
- indvars – pre každý cyklus sa prevedú nasledovné zmeny, indukčná premenná bude vždy začínať na nule a inkrementovať sa bude vždy o jednotku, je garantované, že prvá *phi* inštrukcia v hlavičke cyklu je indukčná premenná a konečne každá pointrová aritmetika je prevedená do indexácie poľa,
- loop-unroll – ide o jednoduché rozbalenie cyklu, ktoré je možné riadiť ďalšími parametrami priechodu,

- instcombine – optimalizácia algebraicky kombinuje inštrukcie medzi sebou, za účelom zmenšiť ich počet,
- licm – je skratka z anglického *Loop Invariant Code Motion*, úlohou priechodu je presunúť čo najviac inštrukcií z cyklu mimo cyklus, na základe alias analýzy a iných analýz hľadá inštrukcie, ktoré sa nachádzajú v tele cyklu a hodnoty, do ktorých tieto inštrukcie zapisujú sa počas celého výpočtu cyklu nemenia, takéto inštrukcie je možné presunúť pred alebo za cyklus,
- lowerswitch – vyhľadá inštrukciu switch a prevedie ju do vetviacich inštrukcií, ktoré na základe podmienky skočia do jedného alebo druhého základného bloku, táto transformácia pridáva základné bloky a vo svojej podstate robí kód zložitejším,
- inline – priechod kopíruje telá funkcií na miesto ich volania, je nastavená určitá hranica, kedy je použitie výhodné a kedy nie,
- simplifycfg – eliminuje mŕtvy kód a spája základné bloky, ktoré majú vzťah jeden predchodca a jeden nasledovník, odstraňuje tie základné bloky, ktoré nemajú žiadneho predchodcu, eliminuje *phi* inštrukcie bloku, ktorý ma len jedného predchodcu a eliminuje bloky, ktoré obsahujú len jedinú inštrukciu a to inštrukciu nepodmieneného skoku.

2.4 Architektúra Codix–VLIW

V rámci projektu Lissom[2] vznikla architekrúra *Codix–VLIW*. Architeltúra bola namodelovaná v jazyku *Codal* ako rozšírenie RISC architektúry *Codix* popísanej v ďalšej podkapitole.

Architektúra je modelovaná ako určitá šablóna. Je jednoducho meniteľná a rozšíriteľná. Jej variabilita spočíva aj v možnosti menení počtu slotov v dlhom inštrukčnom slote. Keď sa pri preklade zistí, že ani použitím mnou implementovaných optimalizačných techník nie sme schopný pokryť napríklad všetky štyri sloty v dlhom inštrukčnom slove, je možné v modely jednoducho znížiť počet slotov. Z nového modelu vygenerujeme automaticky nástroje a je možné ďalej pokračovať ladením programu. Táto variabilita je jedna z predných vlastností *Codasip toolchain*, že je možné jednoducho zmeniť architektúru podľa požiadavkov užívateľa.

2.4.1 Architektúra Codix

Architektúra je popísaná jazykom *Codal*. Jedná sa o univerzálnu 32–bitovú RISC architektúru [3]. Procesor je rekonfigurovatelný, napríklad je možné pridať alebo odobrať inštrukciu. Ďalej je možné implementovať rôzne rozšírenia.

Jedna zo základných vlastností architektúry *Codix* je, že v hardvéri sa nerieši väčšina hazardov, okrem čakania na výsledok inštrukcie *load* a jednoduchého predania výsledku v inštrukčnej linke (forwarding). Všetky ostatné hazardy musí riešiť prekladač. Základný datový typ je 32–bitový integer a architektúra je kontatibilná s architektúrou *ARM* na úrovni jazyka C. Samotná inštrukčná sada je založená na inštrukčných sadách architektúr *ARM*, *MIPS* a medzikódu *LLVM IR*.

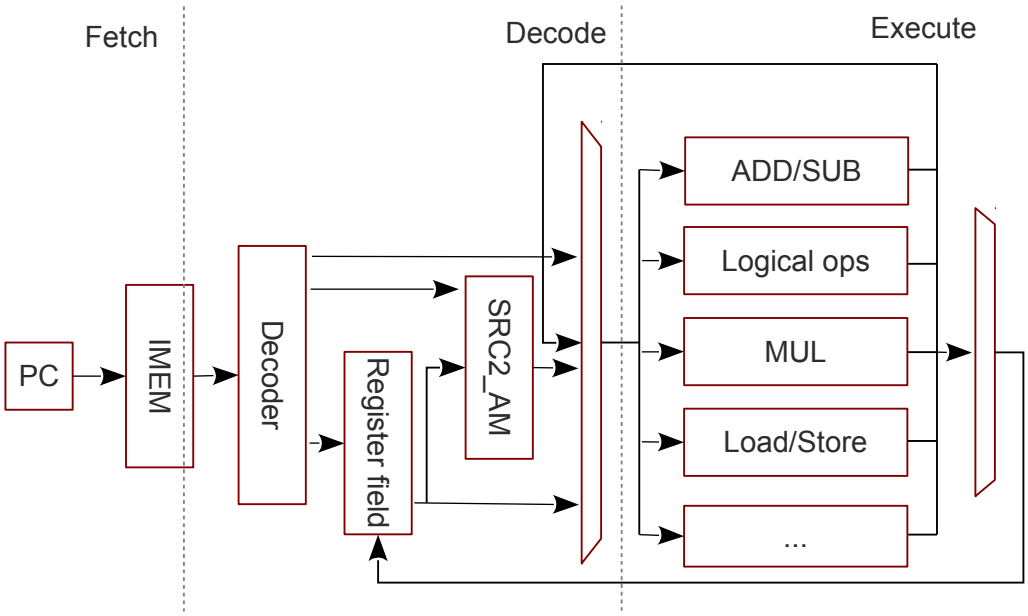
[2] http://www.fit.vutbr.cz/research/groups/lissom/

V základe obsahuje 32 obecných registrov, kde prvý register *r0* obsahuje vždy nulu. Pointer na zásobník je uložený v registry *r1*, návratová hodnota sa ukladá do *r31* a *frame pointer* do *r2*.

Pamäť obsahuje spoločný priestor pre kód aj dáta. Má formu little–endian, 32–bitový adresový priestor a je adresovatelná po 8–bitových slovách. V čase vývoja architektúra neobsahovala inštrukčnú ani dátovú cache pamäť.

Medzi prednosti tejto architektúry patrí jednoduchší hardvér na základe ignorovania väčšiny hazardov. Obsahuje, ale inštrukcie ktoré majú tri vstupné operandy a tiež obsahuje inštrukciu *select*, ktorá je totožná s rovnakou inštrukciou z LLVM IR. Inštrukcia *select* očakáva tri operandy a zapisuje do štvrtého. Prvý parameter je výsledok podmienky, ak tento výsledok je rôzny od nuly do výsledného operandu sa zapíše tretí parameter. Ak naopak výsledok podmienky je rovný nule do výsledného operandu sa zapíše druhý parameter.

Na obrázku 2.6 je uvedená zjednodušená mikro–architektúra. Inštrukčná linka sa skladá



Obrázok 2.6: Codix mikro–achitektúra. Obrázok inšpirovaný z [3]

.

z troch častí. V časti *fetch* sa manipuluje s programovým čítačom a inicializuje sa čítanie z inštrukčnej pamäte.

V časti *decode* sa dokončuje načítanie inštrukcie. Preto je blok *IMEM* zakreslený vo *fetch* aj *decode* časti. Dekóduje sa inštrukcia (*Decoder*) a načítavajú sa hodnoty z registrov. Pomocou bloku *SRC2_AM* je možné druhý operand upraviť (napríklad bitovým posunom).

V časti *execute* sú vytvorené nezávislé dátové cesty pre skupiny operácií. Jednotlivé funkčné jednotky môžu, ale nemusia byť zreťazené. Ich výstup je pripojený späť na registrové pole s omedzením, že len jedná inštrukcia môže v danom cykle zapisovať do registrového poľa. Riešenie tohoto omedzenia je ponechané na prekladači.

Kapitola 3

Metódy a algoritmy optimalizácií pre VLIW

3.1 Súčasný stav

Časť práce už bola spravená v rámci mojej bakalárskej práce [15]. Vtedy bolo do LLVM a programu backengen (generátor backendov z Codasip frameworku) doplnená nutná podpora pre tzv. *bundling*. Táto podpora spočíva v tom, že rôzne architektúry majú rôzny počet funkčných jednotiek a rôzne navrhnutý hardvér. *Bundling* v tomto prípade znamená vytváranie veľmi dlhých inštrukčných slov, tak aby boli korektné a spustiteľné na danej architektúre. Túto funkčnosť som vtedy dosiahol tak, že tesne pred emitáciou kódu v backende LLVM som vložil priechod, ktorý zo základného bloku vytvoril graf závislostí a v rámci tohoto grafu som vyberal tie inštrukcie za pomoci heuristiky *critical path*[1], ktoré mohli byť spustiteľné zároveň (neboli medzi nimi žiadne závislosti) a zároveň funkčné jednotky pre dané operácie boli voľné. Toto rozšírenie síce umožnilo v Codasip frameworku prekladať architektúry typu VLIW, ale výsledný prekladaný program nebol optimálny. Prekladače pre VLIW architektúry pre zvýšenie ILP používajú globálne plánovanie. Globálne plánovanie je technika kedy sa inštrukcie neplánujú a teda nepremiestňujú len v rámci základného bloku, ale v rámci niekoľkých základných blokov zároveň. Tieto zhluky blokov sú vyberané na základe profilu vykonávania programu.

Pri implementácií zadania tejto práce sa v rámci *Codasip* tímu rozhodlo, že prekladač a ostatné nástroje prejdú na novú verziu LLVM a tým implementáciu *bundlingu* z bakalárskej práce bolo nutné implementovať znovu. Nová implementácia využíva novú podporu *bundle* LLVM a implementuje samotný *bunding* novým spôsobom, ktorý bude ďalej v texte popísaný (sekcia 4.4).

3.2 Globálne plánovanie

Zvyšovanie úrovne paralelizmu na úrovni inštrukcií (ILP) plánovaním základných blokov má určité svoje medze. Väčšinou je to počet inštrukcií v typickom základnom bloku, ktorý je často nízky od 5 do 20 inštrukcií. Ak chceme inštrukcie plánovať pre VLIW architektúry a optimálne chceme mať čo najviac slotov v dlhom inštrukčnom slove zaplnených, veľkosť základného bloku nás začne obmedzovať.

[1] Heuristika vyberá z grafu vždy ten uzol, ktorý ma splnené všetky závislosti a zároveň cesta z neho je najdlhšou ku konečnému uzlu.

V tejto kapitole sú uvedené techniky globálneho plánovania, kde je snaha použiť plánovacie algoritmy cez hranice základných blokov. Globálne plánovacie algoritmy, môžme rozdeliť do dvoch skupín:

- metódy založené na profile (profilling methods),
- metódy ovládané štruktúrou (structure–driven methods).

Do metód založených na profile patria algortimy ako *trace scheduling*, *superblock scheduling*, *hyperblock scheduling* a *treeregion scheduling*. Tieto metódy budú popisované v nasledovných podkapitolách.

Metódy ovládané štruktúrou nie sú až tak známe ako metódy založené na profile. Patrí medzi ne napríklad algoritmus *Percolation Scheduling* [16]. Používa štyri transformácie a to *delete*, *move*, *move conditional* a *unify*. Pre každý uzol v CFG sa snaží aplikovať všetky štyri transformácie dookola, dokiaľ je možné aplikovať aspoň jednu z nich. Ďalšie algoritmy, ktoré patria do tejto skupiny sú *Trailblazing*, ktorý je odvodený od *Percolation Scheduling* a *Meld Scheduling*.

3.2.1 Trace Scheduling

Algoritmus je založený na vytvorení *trace*, čo je spôsob ako zlúčiť niekoľko základných blokov. Zverejnil ho prvýkrát pán Fisher v [10]. Za *trace* sa považuje cesta skrze program, ktorá sa skladá z najviac frekventovaných základných blokov. Vytvára sa tak, že sa vezme najfrekventovanejší základný blok a vloží sa do *trace*. Snažíme sa rozširovať *trace* smerom hore k predchodcom vybratého základného bloku a tiež opačným smerom k nasledovníkom vybratého bloku, vyberáme vždy najviac frekventované základné bloky. Vytvárame toľko *trace*, aby každý blok patril práve do jednej *trace*. *Trace* teda môže obsahovať bočné vstupy aj bočné výstupy. Pre každý základný blok v *trace* platí, že *trace* obsahuje len jedného nasledovníka alebo predchodcu.

Na takto vytvorenú *trace* sa použije algoritmus *List Scheduling*, ktorý je primárne určený pre plánovanie inštrukcií v základnom bloku. Pri pohybe inštrukcií cez podmienky základných blokov, je nutné generovať kompenzačný kód.

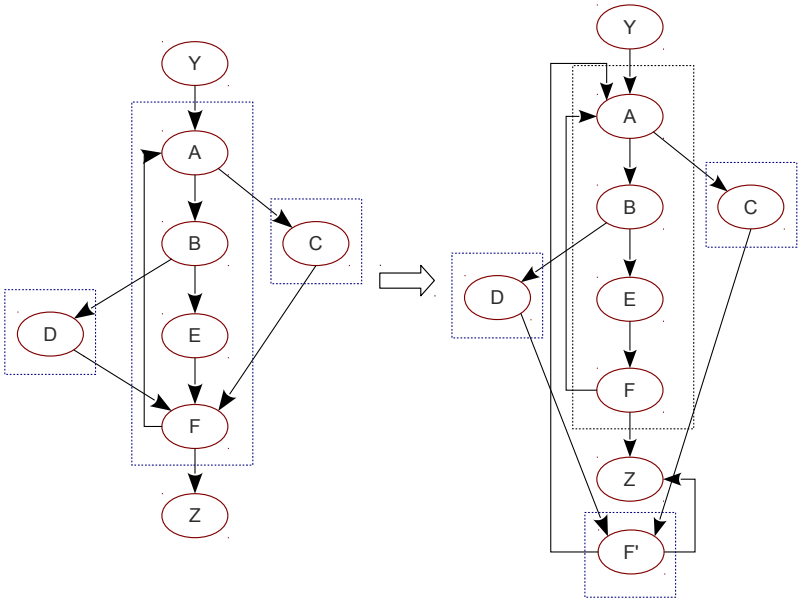
3.2.2 Superblock Scheduling

Superblock je podobný útvar základných blokov ako *trace*, len obsahuje jeden vstup skrze prvú inštrukciu. Výstupov z útvaru môže obsahovať niekoľko. Vytvára sa z *trace* pomocou metódy *tail duplication*. Táto metóda eliminuje postranné vstupy a to tak, že klonuje základné bloky, do ktorých bočný vstup vstupuje. Pre lepšiu predstavu uvádzam obrázok 3.1.

Na obrázku 3.1 je možné vidieť, že základné bloky A a F majú postranné vstupy. Pre základný blok A nemusíme prevádzať zmeny, lebo definícia *superblocku* dovoľuje mať prvému základnému bloku vo formácií postranné vstupy.

Pre základný blok F sa prevedie klonovanie — *tail duplication*. Vytvorí kópiu základného bloku F, v obrázku označená ako F'. Do tejto novej kópie nastavíme postranné vstupy z F. Nakoľko základný blok F' je v *superblocku* jediný je dovolené, aby obsahoval postranné vstupy.

Na *superblok* je možné aplikovať tiež *List Scheduling*, ale rôzne heuristiky pri výbere uzlu ovplyvňujú kvalitu výsledného riešenia. Medzi základné heuristiky s ktorými som sa stretol sú *critical path*, *SR* a *DHASY* (viac v kapitole 3.3).



Obrázok 3.1: Tail duplication. Obrázok je inšpirovaný [14].

3.2.3 Hyperblock

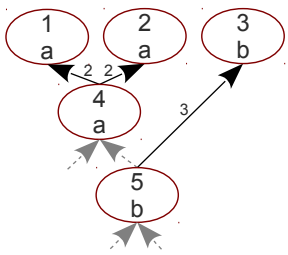
Rovnako ako *Superblock Scheduling* vychádza z *trace*. Bočné výstupy sa snaží eliminovať, ale iným spôsobom. Používa sa predikcia podmienok a čím sa kód linearizuje a základné bloky sa zväčšia.

3.2.4 Treeregion

Treeregion je väčší zhluk základných blokov, ktorý na rozdiel od predchádzajúcich môže obsahovať niekoľko potomkov z jedného základného bloku. Ale každý základný blok nemôže mať viac ako jeden vstup.

3.3 Heuristiky výberu uzlu

V ďalších podkapitolách sú popísané heuristiky výberu uzlu z grafu pri plánovaní inštrukcií napríklad metódou *List Scheduling*. Majme situáciu, ktorá je zobrazená na obrázku 3.2. Obrázok zobrazuje niekoľko inštrukcií a ich závislosti medzi sebou. Číslami sú ozna-



Obrázok 3.2: Acyklický graf závislostí inštrukcií.

čené jednotlivé inštrukcie a písmenom je označený základný blok do ktorého patria. Čiernymi šípkami sú zobrazené dátové závislosti medzi inštrukciami. To znamená, že napríklad inštrukcia *4a* v obrázku je dátovo závislá na inštrukciách *1a* a *2a*. Inštrukcie *4a* a *5b* sú koncové inštrukcie základného bloku a sú to skokové inštrukcie. Spojenie medzi *5b* a *4a* deklaruje, že koncová inštrukcia zo základného bloku *b* nepredbehne koncovú inštrukciu zo základného bloku *a*, teda že sa vykonajú v poradí ako určil programátor. Jednotlivé čísla pri šípkach určujúce závislosti označujú *latenciu* jednotlivých inštrukcií, teda dĺžku ich vykonávania.

V prvom kroku plánovania je možné si vybrať z uzlov *1a, 2a* a *3b*. Podľa heuristiky popísané v nasledujúcom texte určujú, ktorý z týchto uzlov sa v takomto prípade vyberie.

3.3.1 Critical Path

Heuristika *Critical Path* je jedna z najpoužívanejších heuristík pri použití plánovacieho algoritmu *List Scheduling* nad základným blokom. Pre každý uzol je spočítaná najdlhšia cesta z uzlu do konečného uzlu. V príklade z obrázku 3.2 sivé čiarkované hrany majú latenciu 0, takže dĺžka kritickej cesty pre uzol *1a* bude 2. Dĺžka kritickej cesty z uzlu *2a* bude 2 a pre uzol *3b* 3. Použitím tejto heuristiky je snaha kritickú cestu v grafe vždy minimalizovať a to tak, že sa vyberie vždy uzol s najdlhšou kritickou cestou. V našom príklade je to uzol *3b*.

3.3.2 SR

SR je skratka **S**uccessive **R**etirement. Heuristika a jej teoretický základ je dobre popísaný v [7].

Z praktického hľadiska táto heuristika uprednostňuje inštrukcie, ktoré patria do základného bloku, ktorý sa v zdrojovom programe vyskytol skôr. V našom príklade z obrázku 3.2 sa základný blok *a* vyskytuje v zdrojovom programe skôr preto, inštrukcie *1a* a *2a* budú mať vyššiu prioritu.

3.3.3 DHASY

DHASY je skratka **D**ependece **H**eight **a**nd **S**peculative **Y**ield [9]. Táto heuristika je zobecnená heuristika *Critical Path* pre superbloky. Pre každý uzol, ktorý má splnené svoje závislosti to znamená, že inštrukcie na ktorých bol dátovo alebo štrukturálne závislý boli už naplánované, sa spočíta vážený súčet dĺžok kritických ciest ku každej výstupnej hrane zo superbloku. Váha jednotlivých dĺžok je určená podľa pravdepodobnosti daných výstupných hrán.

3.3.4 G*

Táto heuristika (viac informácii v [7]) je viac výpočetne náročná ako predchádzajúce heuristiky. Heuristika sa snaží nájsť kompromis medzi heuristikami *Critical Path* a *Successive Retirement* tým, že aplikuje *SR* na niektoré kritické uzly, z ktorých vedie bočná výstupná hrana.

3.3.5 Speculative Hedge

Podrobný popis a teoretický základ tejto metódy je možné nájsť v [8]. Heuristika uprednostňuje frekventovanejšie uzly, s výstupnými hranami. V podstate každá inštrukcia sa berie

ku vzťahu k výstupným inštrukciám. Inštrukcia, ktorá napomáha ku rýchlejšiemu naplánovaniu frekventovanejšieho uzlu má vyššiu prioritu. Napomáhať môže dvojitým spôsobom, buď leží na kritickej ceste k danému výstupnému uzlu alebo obsadzuje výpočetný zdroj, ktorý daný výstupný uzol potrebuje.

3.3.6 Balance Scheduling

Heuristika priraďuje jednotlivým výstupným uzlom inštrukcie, ktoré je nutné naplánovať skôr ako samotné výstupné uzly. Heuristika ďalej vyberá tie výstupné uzly, ktorých množiny nutných predchádzajúcich uzlov majú čo najväčší prienik. Detailné informácie je možné nájsť v [9].

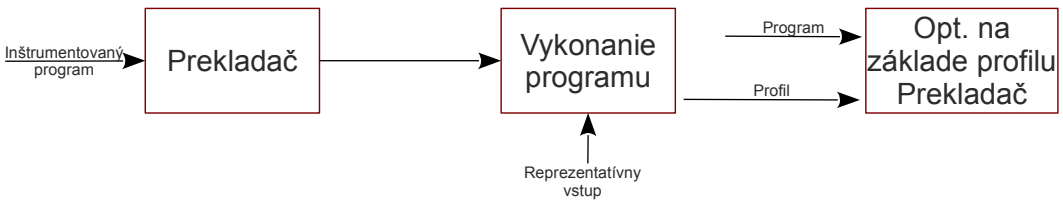
3.4 Získavanie profilu

Nasledujúce informácie sú čerpané z [11].

Klasické optimalizácie, ktoré sú prevádzané v dobe prekladu programu môžu radikálne optimalizovať prekladaný program, ale sú obmedzené tým, že nemajú žiadnu reálnu informáciu o skutočnom behu programu. Túto slabinu sa prekladače snažia zastúpiť pomocou rôznych statických analýz, ktoré majú napomôcť k lepšej aplikácií optimalizácií. Môže sa však stať, že aplikovaním niekoľkých nevhodne zvolených optimalizácií môžme znížiť celkový výkonnosť programu, lebo budú aplikované na nevhodné alebo málo vykonávané časti programu.

Druhý prístup je získanie profilu z behu prekladaného programu a využitie tohoto profilu pre možné optimalizácie. Na základe profilu môžme previesť dva druhy optimalizácií. Prvý druh je aplikovanie optimalizačného priechodu na miesta, ktoré neboli odhalené statickými analýzami a prinesú zvýšenie výkonnosti pre celý program. Druhý druh je aplikovanie optimalizačného priechodu a získanie zvýšenie výkonnosti v jednej časti programu, ale zároveň zníženie výkonnosti v inej časti programu.

Ak chceme získať profil na základe ktorého budeme prevádzať optimalizácie je nutné, zvoliť dostatočne vhodné reprezentatívne vstupné dáta. Je nutné pokryť celú množinu očakávaných vstupných dát, aby sme získali vierohodný profil. Ak by sa získal profil a následne previedli optimalizácie len na základe jedného druhu vstupu, výsledný program v behu s reálnymi dátami by mohol dosahovať nižšieho výkonu.



Obrázok 3.3: Postup prekladu pri použití optimalizácií na základe profilu. Inšpirované [11].

3.4.1 Druhy profilov

Výsledkom profilácie je uloženie informácií o behu programu do profilu. Na základe toho aké informácie sa do profilu ukladajú je možné profily rozdeliť do troch základných skupín

a to profil hodnôt, profil adries a profil toku riadenia programu. V ďalšom texte sa budeme najviac venovať poslednej skupine profilu, lebo práve tento druh profilu je použitý pri riešení.

Profil hodnôt

Tento druh profilu sleduje jednotlivé hodnoty operandov inštrukcií v programe. Vytvára tabuľku, ktorá obsahuje inštrukcie a ich jednotlivé operandy. Ku každému operandu sú priradené dva čísla. Prvé reprezentuje hodnotu operandu (napríklad 5), druhé číslo reprezentuje frekvenciu výskytu danej hodnoty. Takže výsledný záznam môže vyzerať nasledovne.

$$(I1, R3) - (5, 100)$$

Prvá dvojica odpovedá identifikátoru inštrukcie a operandu. Druhá dvojica hodnote operandu a frekvencií výskytu.

Na základe tejto informácie môže prekladač nájsť operandy, ktoré sú konštantné a aplikovať na ne optimalizácie ako propagáciu konštánt a podobne.

Profil adries

Profil adries je najčastejšie reprezentovaný zoznamom adries do pamäte na ktoré bolo počas vykonávania programu pristupované. Programový pamäťový priestor môže byť rozdelený na *zásobník, tzv. hromadu* a *globálny priestor*. Snahou optimalizácií, ktoré sú založené na profilu adries, je zvýšiť výkonnosť programu tým, že sa zefektívni prístup do pamäte. Teda efektívnejšie sa navrhne uloženie jednotlivých operandov.

Profil toku riadenia programu

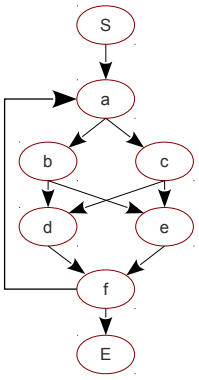
Cieľom tohoto druhu profilu je zachytiť cestu (trace) vykonávania programu. Táto cesta obsahuje poradie jednotlivých základných blokov. Ak základné bloky budú predstavovať uzly a spojíme ich orientovanými hranami, ktoré odpovedajú postupnosti v akej boli vykonávané pri profilácií získame graf, ktorý sa označuje ako **CFG** (control flow graph). Tento druh profilu používajú optimalizácie zamerané na zmenu toku vykonávania programu a presne tieto optimalizácie sú stredobodom tejto práce.

Podľa účelu a spôsobu získania môžme tento druh profilu rozdeliť do nasledovných skupín.

- Profil uzlov – (node profil) pre každý uzol (základný blok) je uložená frekvencia vykonávania.
- Profil hrán – (edge profil) pre každú hranu medzi dvoma uzlami (základnými blokmi) je uložená frekvencia vykonávania.
- Profil dvoch hrán – (two-edge profil) pre každú dvojicu hrán je uložená frekvencia vykonávania. Z tohoto profilu je možné dopočítať *profil hrán*, ale opačne to však možné nie je. Jeho výhodou voči ostatným je možnosť zachytiť vzťah medzi dvomi hranami, ktoré vytvárajú cyklus.
- Profil cesty – (path profil) ukladá sa frekvencia vykonania jednotlivých necyklických ciest skrze **CFG**.

Obecne platí, že z celkového profilu programu je možné získať profil cesty alebo profil dvoch hrán. Z týchto dvoch profilov je možné získať profil hrán a z profilu hrán je možné získať profil uzlov. Opačný vzťah však neplatí.

Pre lepšiu predstavu 3.4 ilustruje upravený príklad z [11].



Uzol	Frekvencia
S	1
a	100
b	80
c	20
d	20
e	80
f	100
E	1

Hrana	Frekvencia
Sa	1
ab	80
ac, ce	20
bd, df	20
cd	0
be	60
ef	80
fa	99
fe	1

Dve hrany	Frekvencia
Sab	1
abd, bdf	20
abe, bef	60
ace, cef	20
dfE, efE	1
dfa	20
efa	80

Cesta	Frekvencia
SabdfE	1
abdf	19
abef	60
acef	20

Obrázok 3.4: Príklad rôznych druhov profilov. Inšpirované [11].

3.4.2 Práca s profilom v LLVM

Kompilačná platforma LLVM obsahuje podporu pre profiláciu a používanie profilu. Je možné na úrovni medzi–kódu vložiť inštrumentáciu, teda špeciálne inštrukcie (napríklad čítače), ktoré pri každom priechode cez priradený základný blok sa inkrementujú. Takýmto spôsobom je možné zostrojiť tabuľku, kde ku každému základnému bloku bude priradená jeho frekvencia.

Inštrumentáciu je možné vložiť pomocou programu *opt*. Pomocou parametru sa zvolí druh profilu, sú dostupné nasledovné možnosti.

- edge profil – typický edge profil,
- GCOV profil – výsledkom je profil, kompatibilný s nástrojom *gcov*, spadajúci do *GNU Compiler Collection*,
- optimal edge profil – vylepšený edge profil,
- path profil – výsledkom je Ball–Laurus path profil [6].

Inštrumentovaný program je možné interpretovať pomocou nástroju z LLVM *lli* alebo preložiť do jazyku symbolických inštrukcií danej architektúry a použiť simulátor z *Codasip toolchain*. Po ukončení simulácie sa vytvorí súbor *llvmprof.out*. Je to binárny súbor, ktorý obsahuje informácie o profile. Názvy základných blokov sú zviazané s programom v medzi–kóde pred inštrumentáciou.

Ak chceme s profilom pracovať ďalej, je možné ho načítať do objektu *ProfileInfoLoader*. Spolu s programom pred inštrumentáciou je možné s ním ďalej pracovať a využiť ho pre optimalizácie. Detailnejší popis práce s profilom v LLVM je možné nájsť v sekcií o tvorbe superblokov 4.3.3.

3.5 Optimalizácie cyklov

Pre zvýšenie paralelizmu na úrovni inštrukcií (ILP) sa používajú rôzne techniky práce z cyklami. Je väčšina programov svojho výpočetného času strávia práve v cykloch. Táto podkapitola pojednáva o troch základných optimalizáciách, ktoré sa pre cykly používajú.

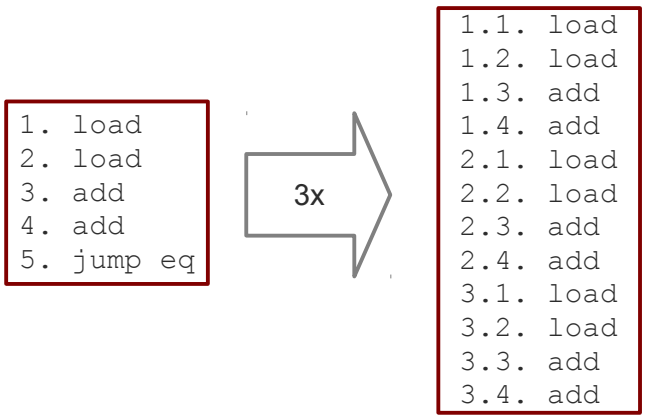
3.5.1 Loop invariant code motion

Je optimalizácia, ktorá síce priamo nezvyšuje ILP, ale napomáha k urýchleniu výpočtu cyklu. Optimalizácia premiestňuje inštrukcie mimo cyklus. Tieto inštrukcie zapisujú do operandov, ktoré sú počas výpočtu cyklu nemenné (tzv. invarianty). Napríklad načítavajú z pamäte nemennú hodnotu. Takúto inštrukciu je možné premiestniť pred samotný cyklus alebo naopak za cyklus.

Táto optimalizácia je už v LLVM implementovaná ako príchod s názvom *licm*.

3.5.2 Rozbalenie cyklu

Majme situáciu zobrazenú na obrázku 3.5 ľavá časť. Obdĺžnik predstavuje jeden základný blok, ktorý obsahuje telo cyklu. Jednotlivé inštrukcie sú očíslované. Prvé inštrukcie načítajú dve hodnoty z pamäte a tretia inštrukcia ich sčíta. Výsledok sčítania sa pričíta do celkového výsledku cyklu a pomocou inštrukcie *jump eq* sa porovná čítač cyklu a skočí sa buď na začiatok alebo sa bude pokračovať ďalej vo vykonávaní programu.



Obrázok 3.5: Úplné rozbalenie cyklu.

Ak by sa mal cyklus v programe vykonať len trikrát, napríklad by sme počítali celkovú sumu dvoch troj–prvkových polí, je možné cyklus úplne rozbaliť ako je zobrazené na obrázku 3.5 pravá časť. Jednoducho sa telo cyklu skopíruje trikrát pod seba v rámci jedného základného bloku. Jednotlivé inštrukcie sú očíslované dvojicou číslic, kde prvé číslo znamená poradie pôvodnej iterácie cykly. Druhé číslo označuje číslo inštrukcie v rámci jednej iterácií.

Takýmto spôsobom získame viac inštrukcií z rôznych iterácií. Pre VLIW architektúry je to výhoda, lebo z pôvodného jednoduchého cyklu sme dostali väčší počet inštrukcií, ktorými môžme efektívnejšie naplniť jednotlivé inštrukčné sloty v inštrukčnom slove.

Jeden z problémov tohoto prístupu je, že rozbaľovať všetky cykly úplne môže priniesť nadmerný nárast kódu, čo nemusí byť chcené. Preto sa používa ešte modifikácia tohoto prístupu tzv. čiastočné rozbalenie cyklu (loop peeling). Ak vieme, že cyklus sa má vykonať napríklad dvadsaťkrát, rozbalíme cyklus len päťkrát a vytvoríme nový cyklus, ktorý sa bude opakovať už len štyrikrát. V príklade z obrázku 3.5 by sa na koniec základného bloku nechala inštrukcia *jump eq* z upravenou koncovou podmienkou.

Ďalšou nevýhodou tohoto prístupu je, že nemôžme rozbaľovať cykly, o ktorých nevieme koľkokrát sa budú opakovať. Exitujú prístupy, ktoré využívajú znalosť programátora, ktorý dokáže určiť minimálny počet opakovania cyklu, tým pádom je možné cyklus aspoň čiastočne rozbaliť.

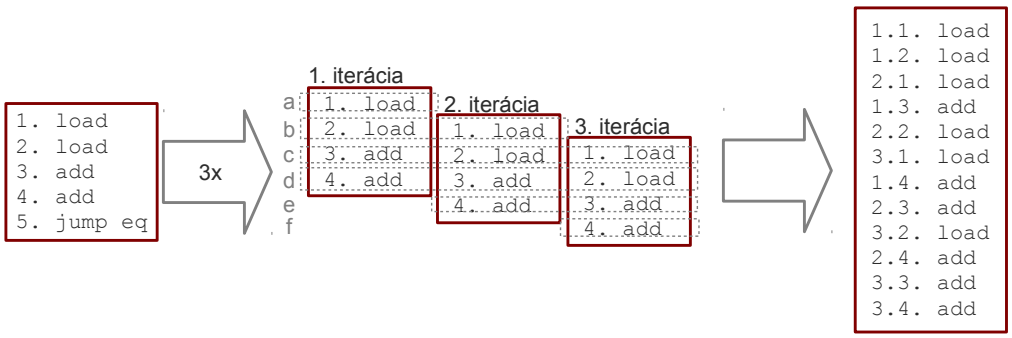
3.5.3 Zreťazenie cyklu

Ďalšou z optimalizácií je zreťazenie cyklu. Rovnako ako rozbalenie cyklov sa táto optimalizácia používa pre zvýšenie ILP. Zakladá na tom, že často jednotlivé iterácie v cykloch nie sú na sebe dátovo závislé. Tým je možné inštrukcie z ďalšej iterácie začať vykonávať ešte počas predchádzajúcej iterácie.

Pre príklad uvádzam obrázok 3.6. Úplne vľavo na obrázku je základný blok s telom jednoduchého cyklu. Majme situáciu, že rovnako ako v predchádzajúcom príklade by sa celý cyklus mal opakovať v pôvodnom programe trikrát. Zreťazenie si môžme predstaviť tak, že jednotlivé iterácie cyklu položíme vedľa seba a oneskoríme o niekoľko inštrukcií (v našom príklade o jednu). Výsledné zreťazené telo bude vyzerať ako na obrázku úplne vpravo. Jednotlivé inštrukcie sú očíslované dvojicou číslic. Prvá číslica značí číslo iterácie, do ktorej inštrukcia patrila a druhé číslo značí číslo inštrukcie v danej iterácií.

Uprostred obrázku 3.6 je niekoľko iterácií položených vedľa seba. Sústredíme sa na jednotlivé obdĺžniky zobrazené čiarkovanou čiarou označené malými písmenami. V prvom obdĺžniku je umiestnená jedna inštrukcia. V obdĺžniku b dve a v obdĺžniku d,c tri inštrukcie. Keby sme mali architektúru VLIW, ktorá by mala možnosť do svojich slotov vložiť dve load inštrukcie a dve inštrukcie sčítania, mohli by sme jednotlivé obdĺžniky naplánovať ako jedno veľmi dlhé inštrukčné slovo. Čiže výsledný program pre architektúru VLIW by mohol vyzerať tak, že v prvom cykle sa spracuje veľmi dlhá inštrukcia a, potom b,c, ... Celý cyklus by sa vykonal v 6 cykloch procesoru, čo je nezanedbateľná úspora. Z hľadiska terminológie sa cykly a a b nazývajú prolog. Cykly c a d sa nazývajú jadro a cykly e a f f epilog. Pri zreťazení je chcené, aby jadro bolo čo najväčšie teda, aby sa mohol zanedbať prolog a epilog.

Nevýhoda tohoto prístupu je, že nie je použiteľný pre cykly, ktoré majú svoje iterácie voči sebe datovo závislé. Tak isto tento prístup nie je možné použiť na cykly, ktoré nemajú pevný počet iterácií. Viac o teoretickom základe a ďalšie informácie o použití tejto techniky pre VLIW architektúry je možné nájsť v [12].



Obrázok 3.6: Úplné zreťazenie cyklu.

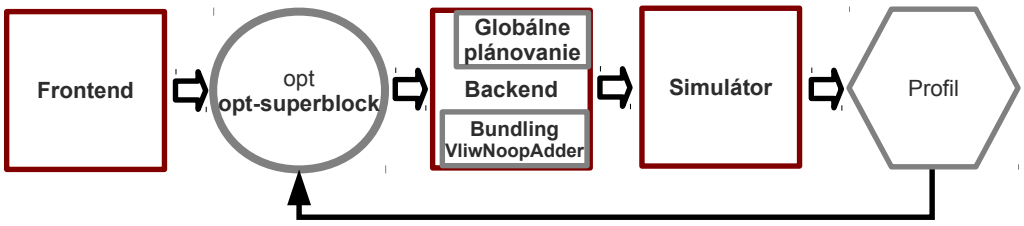
Kapitola 4

Návrh a implementácia

Mojou úlohou je rozšíriť, doplniť a vytvoriť podporu pre možnosť generovať prekladač v *Codasip* frameworku pre VLIW architektúry modelované v jazyku CodAl. Nakoľko generátor používa kompilačnú platformu LLVM, ide o doplnenie a úpravu LLVM.

4.1 Návrh funkcionality

Obrázok 4.1 ilustruje postup prekladu pre uplatnenie profilom riadených optimalizácií. Najprv sa preloží program pomocou frontendu (napríklad program `clang`). Potom sa pou-



Obrázok 4.1: Postup prekladu.

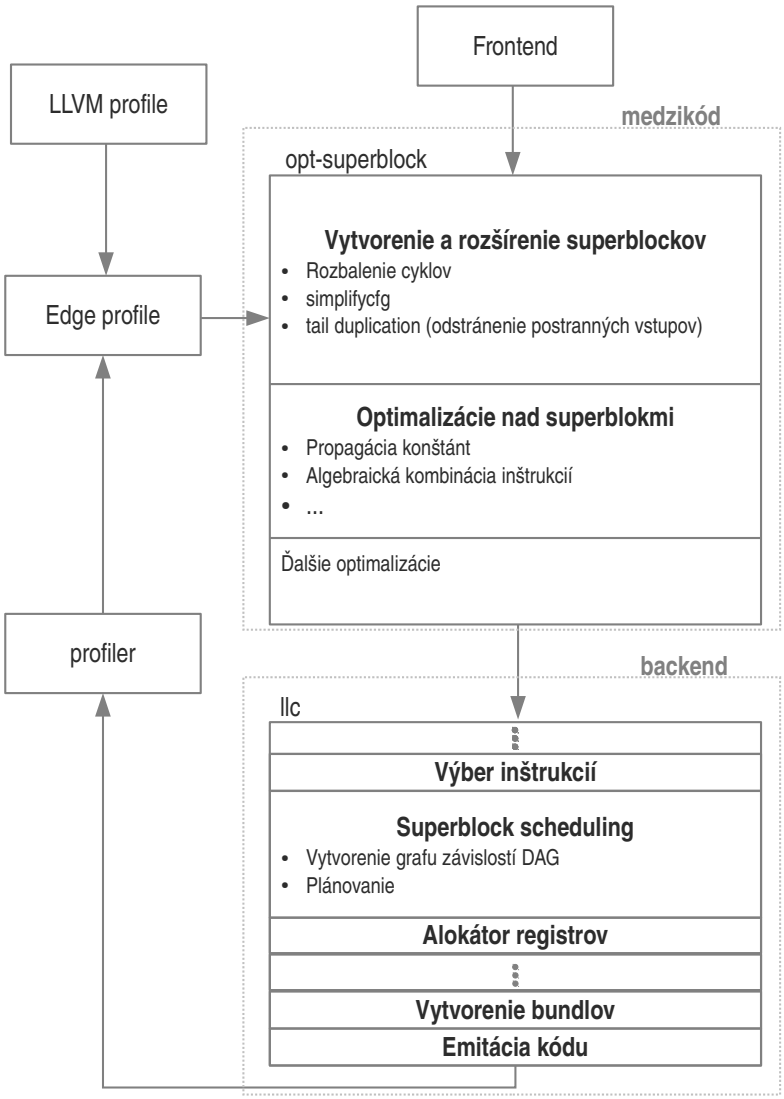
žijú optimalizácie nad medzikódom a tiež sa vloží inštrumentácia. Nasleduje preklad pomocou backendu a simulácia, ktorá vytvorí súbor s profilom. Profil sa načíta program *opt–superblock* na úrovni medzikódu, pomocou programu sa vytvoria superbloky a kód sa znovu preloží.

4.2 Návrh implementácie

Návrh implementácie som sa snažil ilustrovať na detailnejšom obrázku 4.2. Návrh pozostáva vo vytvorení programu *opt–superblock*, ktorý bude možné pustiť v dvoch režimoch. V prvom bez profilu v prvej iterácií prekladu, kde sa použijú optimalizačné priechody z LLVM a hlavne rozbalenie cyklov. Výsledok majú byť čo najväčšie základné bloky.

Druhý režim bude pustenie programu spolu s *Edge* profilom a vytvorenie superblokov. Nad vytvorenými superblokmi bude možné previesť ďalšie optimalizácie.

Do backendu návrh pridáva nový priechod pre plánovanie inštrukcií, ktorý bude schopný plánovať inštrukcie nad väčšími úsekmi kódu ako sú základné bloky.



Obrázok 4.2: Ilustrácia návrhu implementácie.

4.3 Vytvorenie superblokov

Ako bolo už v predchádzajúcej kapitole avizované superbloky sa vytvárajú nad medzikódom. Pre ich vytvorenie som implementoval program *opt-superblock*, ktorého zdrojové kódy sú umiestnene v adresárovej štruktúre LLVM *llvm-3.2/tools/opt-superblock/* (všetky odkazy na zdrojové kódy uvedené v tejto sekcií sú umiestnene v na tomto mieste). Jednotlivé priechody budú ďalej v texte popisované. Čo sa týka programovej štruktúry priechody sú implementované v samostatných súboroch ako *Function* alebo *Module* priechod. V jazyku LLVM to znamená, že priechod sa prevedie buď nad každou funkciou, alebo nad každým modulom. Modul je najvyššia úroveň, nad ktorou je možné v LLVM s kódom pracovať. Najnižšia úroveň je úroveň samotných inštrukcií, resp. jednotlivých operandov.

Program *opt-superblock* môže pracovať v dvoch režimoch. Prvý režim pripraví preklad kód pre vytvorenie superblokov, tým že prevedie vybraté optimalizácie nad cyklami.

Potom sa program ukončí a predpokladá sa vloženie inštrumentácie, preloženie programu a spustenie simulácie, čím sa získa profil. Kód pred inštrumentáciou a profil je vstupom druhého režimu programu *opt-superblock*. Na základe profilu sa vytvoria superbloky a následne sa prevedie niekoľko optimalizácií. Výstupom je program v medzikóde s vytvorenými superblokmi, ktorý sa ďalej preloží a spustí.

Pri testovaní sa časom ukázalo, že prevádzať optimalizácie nad celým kódom, to znamená nad všetkými modulmi a funkciami pri väčších programoch môže viesť ku spomaleniu celého programu, resp. neúmernému nárastu kódu. Napríklad pre aplikáciu *mpeg4* rozbalenie cyklov nad celým kódom môže viesť až šesťkrát väčšiemu kódu. Teda ak kód má v textovej forme medzikódu 34 MB, rozbalený kód môže mať približne až 204 MB, záleží od úrovni rozbalenia. Štatisticky sa časť kódu vôbec neprevedie alebo prevedie len v niekoľkých iteráciách a zbytok výpočtu sa strávi v niekoľkých funkciách, ktoré prevádzajú samotné spracovanie vstupných dát, napríklad spracovanie obrázku v prípade *mpeg4*. Z tohoto dôvodu je dobré prevádzať optimalizácie len nad vybranými funkciami, v ktorých simulátor pri prevádzaní programu trávi najviac času. Tieto informácie je možné získať z *Codasip–simulátora*, keď ho v *Codasip-Studiu* vygenerujeme so zapnutou voľbou *Enable profiling support* a označenou voľbou *C language level* v nastavení hardvérového projektu. Získaný profil zo simulátoru je iný profil, ako ten čo sa používa pri vytvorení superblokov (presný návod na použitie nástrojov sa nachádza v sekcií 4.7). Tento profil obsahuje rôzne informácie o behu programu, ale hlavne obsahuje zoznam inštrukcií, v ktorých simulácia strávila najviac času. Tieto funkcie sú pre nás zaujímavé ako potencionálne funkcie, nad ktorými sa budú prevádzať optimalizácie. Z tohoto dôvodu dokáže program *opt-superblock* prevádzať vybrané optimalizácie len nad niektorými funkciami.

4.3.1 Rozhranie programu

Ovládanie programu som implementoval pomocou povinných a voliteľných argumentov programu. Je možné použiť nasledovné prepínače.

- -i – povinný parameter označujúci vstupný súbor,
- -o – nepovinný parameter označujúci výstupný súbor, ak parameter nie je zadaný výstupný súbor bude mať názov vstupného súboru doplneného o suffix `.out.ll`,
- -s – nepovinný parameter, ktorý označuje, v ktorom móde sa program má spustiť, ak nie je zadaný spustí sa v prvom móde prípravy kódu pre simuláciu a získanie profilu, ak je zadaný spustí sa program v druhom móde, teda vytvorenie superblokov a optimalizácie nad nimi,
- -p – parameter, ktorý označuje súbor s profilom, má zmysel ho použiť len v kombinácií s parametrom *-s*, teda pri vytváraní superblokov,
- -f – nepovinný parameter, ktorý označuje súbor so zoznamom funkcií, nad ktorými má zmysel optimalizácie prevádzať,
- -u – nepovinný parameter, ktorý má zmysel len ak je program pustený bez *-s*, udáva uživateľom definovanú hranicu pre rozbalenie cyklov,
- -l – nepovinný parameter, ktorý je použiteľný len v prípade, kedy je program spustený s parametrom *-s*, udáva spodnú hranicu frekvencie danej hrany, kedy je možné základný blok priradiť do superbloku, alebo nie.

V prípade parametru *-f* sa očakáva, že súbor so zoznamom funkcií bude obsahovať mená funkcií. Vždy na jednom riadku je uvedené jedno meno funkcie, bez akejkoľvek čiarky alebo iného oddeľovača. Ukážka na obrázku 4.3.

```
yuv2rgb_c_24_rgb
put_pixels8_8_c
h264_v_loop_filter_chroma_8_c
decode_residual
check_mv
memmove
```

Obrázok 4.3: Ukážka formátu zápisu zoznamu funkcií pre *opt-superblock*.

Program po spustení vždy pred aplikáciou každého priechodu, uloží aktuálny kód do súboru. Súbory sú označené buď číslom alebo názvom priechodu. Takže je možné sledovať zmeny, ktoré vznikli dôsledkom použitia jednotlivých priechodov.

4.3.2 Príprava pre superbloky

Keď spustíme program *opt-superblock* bez parametru *-s*, nad vstupným kódom sa prevedie niekoľko optimalizačných priechodov, ktoré pripravia vstupný kód pre tvorbu superblokov. Je možné, že vstupný program je už nejak optimalizovaný a tiež nemusí byť, preto doporučujem kód predpripraviť pre tvorbu superblokov týmto spôsobom. Problémy, ktoré môžu vzniknúť sú uvedené v sekcii 4.3.4, ktorá pojednáva o obmedzeniach pri tvorbe superblokov.

Nasleduje zoznam použitých priechodov v poradí, v ktorom sú na vstupný kód aplikované. Funkcionalita priechodov bola popísaná v sekcií 2.3.2.

1. mem2reg
2. loop-simplify
3. loop-rotate
4. lcssa
5. indvars
6. loop-unroll
7. licm
8. simplifycfg
9. lowerswitch
10. simplifycfg
11. inline
12. simplifycfg
13. reg2mem

Základnou myšlienkou prípravy pre tvorbu superblokov je rozbaliť cykly. Na to, aby bolo možné cykly dobre detekovať a rozbaliť ich, je doporučená určitá transformácia cyklov do rovnakého tvaru. To by mala zabezpečiť postupnosť priechodov 1 až 5.

Priechod *licm* je umiestnení po rozbalení cyklu. Tento priechod sa neskoršie púšťa pri preklade *backendu* znovu. Problém, ktorý tu vzniká je, že často tento priechod narušoval štruktúru superblokov a to tým, že priechod vytvorí nový základný blok a predradí ho pred cyklus. Ak cyklus patrí do superbloku a prvý základný blok má viac vstupov, priechod mu predradí ešte jeden základný blok, ktorý ale nesie značku superbloku z cyklu. Tým sa poškodí superblok. Lebo prvý základný blok v superbloku je novo vytvorený základný blok, ale hneď druhý základný blok má niekoľko postranných bočných vstupov, čo je v rozpore s definíciou základných blokov. Toto je jeden z dôvodov, prečo je lepšie túto transformáciu previesť už teraz, aby neskoršie nemala až taký dopad, aj keď ďalšie implementácia v backende počíta s tým, že superbloky môžu byť poškodené.

Priechody *simplifycfg* sú umiestnené po niekoľkých priechodoch za účelom optimalizácie a zjednodušení CFG po aplikovanej transformácií.

Vytváranie superblokov počíta s tým, že vstupný kód bude zbavený inštrukcií *switch*, ktoré značne komplikujú klonovanie základných blokov. Ďalším pomocným dôvodom je, že *Codasip framework* nedisponuje žiadnou procesorovou architektúrou, ktorá by obsahovala inštrukciu *switch*, čiže pri preklade v backende sa táto inštrukcia tak či tak transformuje na *branch* inštrukcie.

Priechod *inline* je umiestnený ako predposledný v rade priechodov. Je to z toho dôvodu, že *inlining* funkcií je omedzený určitou hranicou a domnievam sa, že je lepšie keď sa najprv cykly rozbalia v rámci svojich pôvodných funkcií, kde hranica veľkosti kódu po rozbalení nebude zaťažená kódom volajúcej funkcie a až potom sa vložia na miesto volania. Tým získame väčšie základné bloky, čo je jeden z našich cieľov. Na druhej strane testovanie ukázalo, že príliš veľké funkcie nemusia byť vždy rozumné riešenie (viac o tomto probléme v sekcií 4.7.1).

Ako posledný priechod je umiestnení priechod prevodu registrových operandov do pamäte. Tento priechod spomalí výsledný prekladaný program, ale jeho prínosom je, že odstráni *phi* inštrukcie, čo značne uľahčí klonovanie základných blokov, pri tvorbe superblokov. Klonovanie základných blokov v programe, ktorý obsahuje *phi* inštrukcie nie je triviálna záležitosť, lebo pri vyklonovaní jedného základného bloku, musíte upraviť a skontrolovať inštrukcie v rámci základného bloku, prepojiť ho správne z predchodcom a potomkom, skontrolovať ich inštrukcie. Tým, že v novom základnom bloku vznikli nové inštrukcie a teda tým aj nové hodnoty, je nutné v podstrome programu, ktorý je napojený na klonovaný základný blok rozhodnúť v každom základnom bloku či sa má použiť originálna hodnota alebo nová klonovaná hodnota inštrukcie. Ak program obsahuje cykly, priechod grafom sa stáva komplikovanejší a úprava hodnôt zložitejšia. Situácia sa značne zjednoduší, keď každá hodnota musí byť uložená v pamäti a každý základný blok si ju musí najprv načítať, použiť a uložiť. Tým sa odstránia dátové závislosti medzi základnými blokmi a samotné klonovanie sa značne zjednodušuje. Túto činnosť presne robí priechod *reg2mem* tým, že uloží každú hodnotu do pamäte, odstráni *phi* inštrukcie a teda aj dátové závislosti medzi základnými blokmi.

4.3.3 Vytvorenie superblokov a optimalizácie nad nimi

Spustenie programu *opt-superblock* v druhom režime predpokladá, že sme do výsledného kódu prvého pustenia programu *opt-superbock* vložili inštrumentáciu, preložili, odsimulo-

vali a získali súbor s profilom. Keď máme súbor s profilom je možné pokračovať ďalej v optimalizáciách.

Pustíme program *opt-superblock* s parametrom *-s* a nastavenou cestou ku súboru s profilom cez parameter *-p*. Ako vstupný súbor musíme použiť výsledok prvého pustenia program *opt-superblock*, inak profil nemusí pasovať na vstupný súbor.

Nasleduje zoznam priechodov, ktoré sa aplikujú na vstupný kód s popisom činnosti a odkazmi priamo do zdrojového kódu, nakoľko väčšinu použitých priechodov som implementoval.

Načítanie profilu a vytvorenie superblokov

Pred zavolaním tohoto priechodu sa načíta profil zo súboru do objektu *ProfileInfoLoader* a predá do priechodu ako parameter v konštruktore. Priechod je implementovaný ako LLVM *ModulePass*, čiže nižšie popísaná činnosť sa prevedie pre každý modul vstupného programu.

ModulePass priechod je implementovaný ako trieda, ktorá dedí po nadriadenej triede *ModulePass* a keďže chceme pracovať s profilom je dobré dediť aj po triede *ProfileInfo*.

Pre triedu *ModulePass* je nutné implementovať virtuálnu metódu *runOnModule()*, ktorou sa začína vykonávanie priechodu nad modulom. Ďalšou virtuálnou metódou je *getAnalysisUsage()*, kde je zoznam potrebných analýz, ktoré sa majú vykonať pred spusteným priechodu. V našom prípade potrebujeme informácie o cykloch *LoopInfo*.

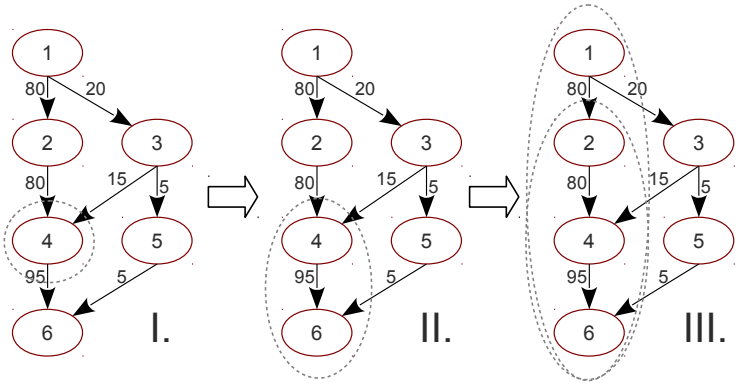
Pre triedu *ProfileInfo* je nutné implementovať virtuálne metódy *recursiveBasicBlock()* a *readEdge()*, ktoré slúžia pre načítanie profilu.

Vykonávanie priechodu sa začína vo funkcií *runOnModule()*. Ako prvá vec je potrebné pre daný modul získať profil, čo implementuje metóda *prepareEdgeInformation()*. Jej hlavná funkčnosť spočíva v prejdení cez jednotlivé základné bloky a vyčítať z profilu informácie o hranách medzi jednotlivými základnými blokmi. Ak použijeme pri inštrumentácií parameter *-insert-optimal-edge-profiling*, nie každá hrana medzi dvoma základnými blokmi musí mať nutne priradený čítač vykonania v profilom súbore. Ak je možné počet vykonania dopočítať, hrana nepotrebuje inštrumentáciu. Cieľom je naplniť mapu *EdgeInformation*, ktorá v tomto prípade mapuje pointer na funkciu na ďalšiu mapu, ktorá priraďuje jednotlivým hranám počet ich vykonania. Deklaráciu je možné nájsť v súbore *llvm-3.2/include/LLVM/Analysis/ProfileInfo.h*.

Keď je profil hrán spracovaný je možné z neho vypočítať *BasicBlock profil*. Počet koľkokrát bol základný blok vykonaný v programe je možné získať aj s profilového súboru, ale pri vývoji som sa stretol s problémami, ktoré som vyriešil tým, že som si profil dopočítal na základe súčtu všetkých hrán vychádzajúcich zo základného bloku. Tento výpočet sa nachádza vo funkcií *countFreqForAllBB()*, kde sa nachádza ešte jeden dôležitý výpočet. Pri tvorbe superblokov sa nie vždy vyplatí pripájať do superbloku každý základný blok. Ideálny prípad je, keď do superbloku patria len tie základné bloky, ktoré boli často vykonávané. Tiež vytvárať superbloky so základných blokov, ktoré boli vykonané len párkrát a kvôli nim zväčšovať celý kód, kvôli procesu *tail duplication* nie je chcené. Preto sa v metóde nachádza výpočet histogramu. Ku každému počtu vykonania danej hrany sa priradí výskyt tohoto počtu v rámci celého modulu. Čím získame informáciu koľko hrán v danom module má rovnaký počet vykonania. Váženým priemerom sa z tohoto histogramu získa spodná hranica, ktorá sa použije pri vytváraní superblokov. Základné bloky, ktoré budú pripojené do superbloku hranou, ktorej počet vykonania je menší ako vypočítaná hranica, sa do superbloku nepridá. Užívateľ môže tiež túto hranicu nastaviť pomocou parametru programu. Ak je hodnota užívateľom nastavená, tak výpočet sa zahodí. Užívateľ musí zadať hodnotu

vyššiu ako 0.

Ďalším krokom pre vytvorenie superblokov je nájdenie tzv. *traces*, ide o rovnaký útvar nad základnými blokmi ako je popísaný v sekcií o *Trace Schedulingu* 3.2.1. Implementáciu je možné nájsť v metóde *createTracesForFunction()*. Na začiatku sa pomocou metódy *chooseSeed()* vyberie základný blok, ktorý bol podľa profilu najviac vykonávaný. Metóda zabezpečuje, že ak sa raz už jeden základný blok vybral, nemôže sa vybrať znovu. Vybratý základný blok sa prehlási za *trace* a novovzniknutú *trace* sa snažíme zväčšovať smerom k predchodcom aj nasledovníkom základného bloku. Zo všetkých nasledovníkov posledného základného bloku sa z *trace* vyberie ten základný blok, ktorý je najlepší. To znamená, že hrana, ktorá ho spája s *trace* má priradenú najvyššiu frekvenciu z pomedzi ostatných potomkov. Aby sme vybraného potomka mohli vložiť do *trace* musí byť jeho predchodca z *trace* jeho najlepší predchodca. Okrem spomínanej podmienky je tu ďalšie obmedzenie a to také, že základný blok, ktorý chceme priradiť do *trace* patrí do rovnakého cyklu ako zbytok *trace*. Celú situáciu ilustruje obrázok 4.4. V prvej fázy sa vyberie najfrekventovanejší



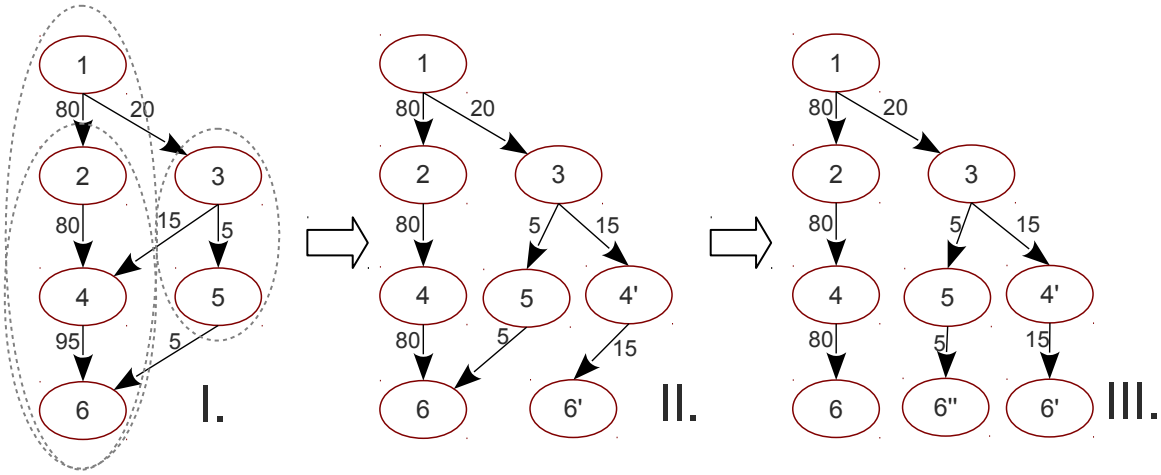
Obrázok 4.4: Ukážka tvorenia *trace*.

základný blok, čo naznačuje sivá elipsa okolo základného bloku číslo štyri. V druhej fázy je snaha rozšíriť *trace* smerom k nasledovníkom. Pridá sa základný blok s číslom 6, nakoľko je to najlepší potomok bloku 4, blok 4 je najlepší predchodca bloku 6 a obidva bloky patria do rovnakého cyklu. V tomto prípade do žiadneho cyklu. V tretej fázy pridáme do *trace* postupne bloky s číslom 2 a 1. Nakoľko blok 2 je najlepší predchodca bloku 4 a blok 1 je najlepší predchodca bloku 4. A tiež platí, že blok 4 je najlepší nasledovník bloku 2 a blok 2 je najlepší nasledovník bloku 1. Ak by bola nastavená hranica obmedzujúca pridania bloku do *trace* napríklad na 85 vo výslednom *trace* by boli len bloky 4 a 6.

Keď máme vytvorené *traces* môžme pokračovať v tvorbe superblokov a to tým, že prejdeme skrze *trace* a odstránime bočné vstupy pomocou metódy *tial duplication*, teda klonovaním základných blokov. Detekcia bočných vstupov je implementovaná v metóde *checkPredecessors()* a odstránenie postranných vstupov v *tailDuplication()*. Je dôležité poznamenať, že bočné vstupy sa testujú na všetky bloky v *trace* okrem prvého, ktorý môže mať neobmedzený počet vstupov. Samotná detekcia prebieha tak, že sa pre každý blok z *trace* testuje či jeho predchodcovia sú z *trace*. Nakoľko je možná situácia, že jeden blok z *trace* môže mať dvoch predchodcov a obidva môžu byť z rovnakej *trace*, táto situácia sa považuje tiež za bočný vstup a je nutné ju ošetriť.

Samotné odstránenie postranných vstupov prebieha nasledovne. Metóda *tailDuplication()* ako parametre preberie odkazy na predchodcu bloku, ktorý sa bude klonovať z *trace*, sa-

motný blok, ktorý sa bude klonovať ako prvý a odkaz na základný blok, ku ktorému sa pripojí vyklonovaná vetva programu. Majme nasledovnú situáciu zobrazenú na obrázku 4.5. Zistili sme, že základný blok číslo 4 má postranný vstup, ktorý je mimo domovskej



Obrázok 4.5: Ukážka tail duplication.

trace. Do metódy *tailDuplication()* teda potrebujeme odkaz na blok číslo 2, samotný blok číslo 4 a blok číslo 3. Ako náhle je nájdený blok s postranným vstupom je nutné klonovať nie len jeho samotného, ale všetky bloky na neho naviazané z danej *trace*. To znamená, že v obrázku sa najprv klonuje blok číslo 4 do bloku $4'$. Tento prvý blok odpojíme od bloku číslo 2 a pripojíme k bloku číslo 3. Tak isto sa musí odpojiť blok 4 od bloku 3. Keby prvý klonovaný blok mal viac predchodcov, všetci sa prepoja s klonovaným blokom (tento proces sa prevedie, len pre prvý klonovaný blok). Pokračujeme v klonovaní bloku číslo 6 na blok $6'$. Nový blok $6'$ sa musí odpojiť od 4 a pripojiť k $4'$. Klonované bloky vytvoria v podstate ďalšiu novovzniknutú *trace*, ktorá sa bude následne testovať na bočné vstupy. V tomto prípade by bolo dobré do *trace* pridať aj blok 3, ale tento prípad nemusí byť pravidlom. Klonovaný blok $4'$ môže mať viac predchodcov. Výsledok klonovania je označený rímskou III. Vznikli dva superbloky, prvý je $1, 2, 4, 6$ a druhý $4', 6'$.

Po vytvorení superblokov, superbloky sú zatiaľ uložené v poradí vo vektore *traces*, prichádza na rad zápis informácie o superblokoch do zdrojového kódu prekladaného programu. V podstate je potrebné každý základný blok, ktorý patrí do superbloku nejakým spôsobom označiť. Toto označenie, by malo byť permanentné, aby pri načítaní blokov v backende bolo možné superbloky identifikovať. Skúšal som rôzne možnosti ako je uloženie informácie do metadát, vytvorenie špeciálnej inštrukcie ... Nakoniec som zostal pri zmene mena základného bloku. Tento spôsob nevyžadoval žiaden zásah do ostatného LLVM a spolieham sa na to, že LLVM nevygeneruje značku a číslo superbloku ako meno alebo časť mena základného bloku. Ak teda základný blok patrí do superbloku doplní sa na koniec mena bloku `__SP__#num__`, kde namiesto `#num` je uvedené číslo superbloku, do ktorého základný blok patrí.

Po vytvorení superblokov nedoporučujem používať optimalizačné priechody, ktoré menia CFG. Respektíve odstraňovať základné bloky je možné, ale tvorba a pridanie nových základných blokov by mohlo mať deštruktívny dopad na vzniklé superbloky.

Abstrakcia nad superblokmi

Superblok je možné implementovať ako zoznam odkazov na základné bloky (v C++ je možné použiť *vector* z *std* knižnice). Deklaráciu a definíciu potrebných metód je možné nájsť v súbore *SuperBlock.h*. Abstrakcia nad superblokmi nachádzajúca sa v programe je implementovaná v súboroch *BunchOfSuperBlocks.h* a *BunchOfSuperBlocks.cpp*. Tiež obsahujú dôležité metódy pre zápis a načítanie superblokov do alebo z programu. Ak máme program, ktorého základné bloky nesú značku superbloku, môžme vytvoriť objekt *BunchOfSuperBlocks* a použiť nad modulom alebo funkciou metódu *readModule()* alebo *readFunction()*, ktorá naplní objekt superblokmi.

Mem2reg a propagácia konštánt

Po načítaní profilu a vytvorení superblokov nasledujú dva LLVM priechody. Prvý vráti kód do formy s *phi* inštrukciami a dátovými závislosťami medzi základnými blokmi. Je nutné poznamenať, že ak na program v medzikóde aplikujeme v poradí priechody *reg2mem* a *mem2reg* nezískame rovnaký kód ako na začiatku. Vo výslednom kóde sa zvýši počet základných blokov. Tieto nové základné bloky budú ako keby navyše. Budú obsahovať často jednu skokovú inštrukciu. Je to daň, ktorú platím za jednoduchšie klonovanie pri tvorbe superblokov. Tento nežiadúci účinok sa snažím odstrániť ďalšími priechodmi.

Ďalším LLVM priechodom je propagácia konštánt. Mohlo sa stať, že zmenou CFG programu je možné túto optimalizáciu použiť. Ďalšou optimalizáciou pri tvorbe superblokov, ktorá sa ponúka pre použitie na základe zmeny CFG je kombinácia redundantných inštrukcií. Pri testovaní sa však ukázalo, že priechod pravdepodobne obsahuje chybu a po jeho použití prestal byť funkčný benchmark *blowfish* z LLVM test suite, preto je tento priechod vyradený z prekladu.

Eliminácia konštantných podmienok

Pri testovaní a prezeraní si CFG s vytvorenými superblokmi som si všimol, že použitými priechodmi sa vytvorili skokové inštrukcie na konci základných blokov, ktoré majú konštantnú podmienku. Napríklad: `br i1 true, label %exit, label %body`. V takomto prípade je možné inštrukciu vymeniť za skok bez podmienky a celý podstrom *false* vetvy vymazať, ak základné bloky z tohoto podstromu nemajú nejakých iných predchodcov. Pri vymazaní bloku, je nutné prejsť celý podstrom a upraviť prípadne *phi* inštrukcie. Implementácia priechodu sa nachádza v súbore *EliminateConstantCond.cpp*.

Ako ďalší priechod sa prevedie LLVM priechod pre elimináciu mŕtveho kódu, teda *dce – Dead Code Elimination*.

Odstránenie prázdnych základných blokov

Tento priechod odstraňuje pozostatky z kombinácie priechodov *reg2mem* a *mem2reg*. Prevodom vznikajú základné bloky, ktoré obsahujú len jednu inštrukciu a to je inštrukcia absolútneho skoku. Tieto bloky je možné odstrániť, ak majú len jedného predchodcu. Vtedy sa do skokovej inštrukcie v predchodcovi nastaví ako cieľ blok, na ktorý ukazuje prázdny blok. V cieľovom bloku je nutné opraviť *phi* inštrukcie.

Ak by prázdny blok mal viac predchodcov ako jedného, úprava kódu pri odstránení tohoto bloku by musela byť viac sofistikovanejšia ako teraz implementovaná úprava. Súčasná implementácia odstraňuje len tie bloky, ktoré majú jedného predchodcu. Priechod je implementovaný v súbore *EraseEmptyBB.cpp*.

Spojenie základných blokov

Jedná sa o veľmi jednoduchý priechod a používa to čo už systém LLVM má implementované. Priechod prechádza vo funkcií každý základný blok a snaží sa ho spojiť s jeho predchodcom pomocou LLVM funkcie *MergeBlockIntoPredecessor()*. Podmienok, kedy je možné potomka spojiť z jeho predchodcom je niekoľko. Pre základnú predstavu patrí medzi ne, že potomok nemôže mať viac ako jedného predchodcu, nemôže sa zničiť cyklus, ...

4.3.4 Obmedzenia tvorby superblokov

Program *opt-superblock* má dve hlavné úlohy. Rozšíriť základné bloky ako najviac je to možné. K tomu má prispieť rozbalenie cyklov a väčšina vyššie vymenovaných priechodov.

Druhou snahou je vytvoriť superbloky na základe profilu. Teda región základných blokov, ktoré nemajú postranné vstupy, ale môžu mať postranné výstupy. Tieto regióny obsahujú základné bloky u ktorých sa predpokladá, že ich vykonanie bude časté. Prvé obmedzenie spočíva v riešení označenia základného bloku, ktorý patrí do superbloku. Riešenie pomocou zmeny názvu a pridaní špeciálneho reťazca nie je úplné. Pri testovaní sa toto riešenie nepreukázalo ako problémové.

Ďalším obmedzením je, že je nutné aby vstup programu *opt-superblock* s parametrom *-s* bol po aplikovaní priechodu *reg2mem*. A takýto vstup bol preložený, odsimulovaný a získaný profil. Ak by užívateľ chcel previesť ďalšie optimalizácie po prvej fázy prekladu je nutné, aby ako svoj posledný priechod použil práve *reg2mem*, lebo druhá fáza prekladu a pustenie *opt-superblock* v druhom režime s parametrom *-s* s tým počíta a inak je možné prísť k zbytočným chybám pri preklade, ktoré sa ťažko ladia.

Pri testovaní som prišiel na ďalšie obmedzenie. Ak sme *frontendom* preložili program do medzikódu a už pri tomto preklade sme použili optimalizáciu *-O3*, medzikód bude už optimalizovaný a cykly rozbalené. Nie vždy sa teda vyplatí púšťať *opt-supreblock* v prvom móde. Ďalším rozbalením sa môže celková rýchlosť programu zhoršiť. Ak je cyklus vhodne rozbalený ďalšie vynútené rozbalenie pridá riadiace základné bloky do programu, čo môže spomaliť výpočet. Preto ak prekladáme aplikáciu a získavame slabšie výsledky ako by sme očakávali, môže byť zrovna toto dôvod slabších výsledkov. V takých prípadoch treba rozumne voliť aké optimalizačné priechody sa na medzikód pustia. Posledným musí byť *reg2mem*, potom preklad do asembleru, simulácia, získanie profilu a použitie *opt-superblock* v druhom režime s parametrom *-s*.

Problém, ktorý je špecifický pre architektúru *Codix-vliw* a je možné, že sa bude opakovať aj pri ostatných architektúrach, je nasledovný. Architektúra *Codix-vliw* obsahuje inštrukciu podmieneného skoku, ktorá má vyhradené určité miesto pre adresu cieľa. Ak sa kód neúmerne rozbalí, *inlining* spôsobí príliš veľké funkcie, môže sa stať, že vyhradené miesto pre adresu cieľa nemusí pre doskok stačiť. Túto situáciu stráži priechod v *backende* s názvom *JumpLenghtChecker*, bude ďalej popísaný. Popisovanú situáciu vyrieši negáciou podmienky a vložením ďalšieho absolútneho skoku hneď za podmienku. Čiže logika programu zostane zachovaná, ale rýchlosť programu je zaťažená ďalšou skokovou inštrukciou s prázdnym *bundle*, čo v globálnom merítku spomalí výsledný program.

4.4 Vytváranie bundle

Priechod pre vytváranie *bundle* zo zoznamu inštrukcií v základnom bloku je súčasťou backendu. V rámci backendu je možné umiestnovať len priechody nad funkciami. Priechody nad

modulmi nie je možné vytvoriť.

Priechod je umiestnení tesne pred emitáciou výsledného kódu do asembleru. Dôvod je jednoduchý. Nie je chcené, aby sa do kódu po prevedení *bundlingu* pridávali alebo odoberali jednotlivé inštrukcie. Takže je dobré, keď sú už alokované registre a je vložení prológ a epilóg. Keď sa vytvoria *bundle* sú prevedené ešte dva priechody, ktoré budú popísané v ďalšom texte. Ide o *VliwNoopAdder* a *JumpLenghtChecker*, obidva priechody pracujú už nad *bundle*.

Bundling sa prevádza nad jedným základným blokom, čiže nejedná sa o globálnu optimalizáciu. Jeho úlohou je v rámci základného bloku nájsť inštrukcie, ktoré nemajú medzi sebou žiadne ani dátové, ani štrukturálne hazardy a tie umiestniť do jedného *bundle*. Z pohľadu teoretickej informatiky sa tu riešia dva úplne NP problémy. Prvý problém je plánovanie. Z inštrukcií je nutné postaviť graf závislostí a vyberať z grafu vždy optimálnu inštrukciu.

Druhý úplný NP problém je ukrytý v tvorbe *bundle*. Ak by sme mali k dispozícií viac inštrukcií ako je možné vložiť do *bundle* stojíme pred optimalizačnou úlohou, ktorá by sa dala redukovať na *knapsnack* problém. Kedy sa snažíme do batohu vložiť predmety rôznej veľkosti a ceny (je možné pridať ďalšie parametre ako váhu ...) a samozrejme chceme odniesť čo najviac predmetov, zároveň chceme mať batoh čo najnižšej váhy. *Bundle* pre nás môže predstavovať batoh, do ktorého vkladáme inštrukcie, ktoré majú rôznu latenciu, rôzne dovolené pozície v bundle.

4.4.1 Podpora LLVM

LLVM vo verzií 3.2 obsahuje podporu pre tvorbu *bundle*. Táto podpora je založená na tom, že na základe itenerára z *ArchSchedule.td* (jedná sa o jeden zo súborov backendu, *Arch* je zameniteľné za meno architektúry) sa vytvorí konečný automat, kde jeden stav automatu predstavuje aktuálny stav *bundle* a jednotlivé hrany predstavujú vloženie ďalšej inštrukcie do *bundle*. Problém, ktorý tu vzniká je taký, že samotná implementácia v LLVM nerieši vzťahy medzi *bundle*. Ďalší problém vznikol, keď som implementoval priechod, ktorý využíval túto podporu do backendu, tak preklad pre nami napísaný detailný itenerár trval niekoľko minút, čo bolo pre tak jednoduché aplikácie neprípustné. Preto som zobral priechod, ktorý som implementoval vo svojej bakalárskej práci a prepracoval som ho do novej podoby.

Priechod tak ako v minulosti je umiestnený v *llvm-3.2/lib/CodeGen/PreEmitVliwPass.cpp*. Výstupom priechodu je kód obsahujúci LLVM *bundle*. Jedná sa o ďalšiu vec, ktorá do LLVM prišla poslednými verziami. LLVM obsahuje abstrakciu nad *bundle*. Je ju možné vytvoriť nad rôznym počtom inštrukcií v *bundle*. Tieto inštrukcie dostanú nastavený príznak *InsideBundle*. *Bundle* sú potom reprezentované ako jedna špeciálna inštrukcia. Cez jednotlivé *bundle* v rámci základného bloku je možné iterovať pomocou *MachineBasicBlock::iterator*. Ak chceme prechádzať jednotlivé inštrukcie a nebrať *bundle* v úvahu je možné použiť *MachineBasicBlock::instr_iterator*. Tento spôsob spôsobuje, že zápis algoritmov pracujúcich s *bundle* nie je moc prehľadný, preto v ďalších verziách LLVM sa tento prístup zmenil.

4.4.2 Implementácia

Priechod obsahuje definíciu dvoch tried *PreEmitVliwScheduler* a *SchedulePreEmitVliwList*. Prvá trieda dedí od triedy *MachineFunctionPass* a implementuje samotný priechod cez funkciu, základné bloky a inštrukcie. Je dôležité spomenúť spôsob akým sa prechádzajú inštrukcie. Majme nasledujúcu situáciu, obrázok 4.6. Ukážka je z reálnej aplikácie, aby čitateľ mal predstavu ako inštrukcie vyzerajú v čase prekladu tesne pred *bundlingom*. Každý

```
1.  %regs_3<def> = i_ld %regs_2, -20; mem:LD4[%j]
2.  %regs_3<def> = i_ld %regs_4<kill>, %regs_3<kill>; mem:LD4[%arrayidx3]
3.  %regs_4<def> = i_ld %regs_2, -36
4.  i_jump_call_abs__opc_calli__addr26__GA_ <ga:@init_search>,
5.  %regs_3<def> = i_ld %regs_2, -20; mem:LD4[%j]
6.  %regs_4<def> = i_ld %regs_2, -40
7.  %regs_3<def> = i_ld %regs_4<kill>, %regs_3<kill>; mem:LD4[%arrayidx4]
8.  i_jump_call_abs__opc_calli__addr26__GA_ <ga:@strsearch>
9.  i_st %regs_3, %regs_2, -28; mem:ST4[%here]
10. i_jumpc_cond_eq %regs_3<kill>, <BB#6>
```

Obrázok 4.6: Reálna ukážka základného bloku po alokácií registrov.

riadok som ručne očísloval a skrátil mená inštrukcií. *Regs_* sú označené registre, prefixom *i_* začína meno inštrukcie. Informácia za bodkočiarkou u *store (i_st)* a *load (i_ld)* inštrukcií označujú *memory* operandy, ktoré sú dôležite pri stavbe grafu závislostí a alias analýze. Kód v ukážke načíta do registrov hodnoty, zavolá funkciu *init_search*, načíta do registrov ďalšie hodnoty, zavolá funkciu *strsearch*, uloží sa register do pamäte a na za základne hodnoty *regs_3* sa rozhodne či sa prevedie posledný skok alebo nie.

Inštrukcie sa prechádzajú od poslednej inštrukcie a hľadá sa inštrukcia, ktorá predstavuje hranicu pre plánovanie tzv. *SchedulingBoundary*. Medzi takéto inštrukcie patria všetky *branch, call, label* inštrukcie a ďalšie, ktoré užívateľ označí. V príklade z obrázku 4.6 sa bude postupovať od poslednej inštrukcie po ôsmu inštrukciu, ktorá je *call* a teda aj *SchedulingBoundary*. Úsek kódu od inštrukcie deväť (vrátane) po inštrukciu desať sa vezme a pošle do plánovača — bundleru. V ďalšej iterácií sa zoberú inštrukcie od inštrukcie 5 po inštrukciu 8 a v poslednej od 1 po 4 vrátane.

Bundler je implementovaný triedou *SchedulePreEmitVliwList*, ktorá dedí od *ScheduleDAGInstrs*. Najdôležitejšie položky v triede sú *AvailableQueue*, čo je objekt triedy *LatencyPriorityQueue* deklarovaný v súbore *LatencyPriorityQueue.h*. Ide o implementáciu fronty, s podporou heuristiky *critical path*. Do fronty sa ukladajú uzly grafu závislostí inštrukcií, ktoré majú splnené svoje závislosti a je možné ich plánovať. Keď je uzlov viac, je dôležité poradie v akom sa uzly z fronty vyberú. Fronta *AvailableQueue* pri použití metódy *pop* vyberie vždy ten uzol, ktorého cesta v grafe bola najdlhšia.

Detektor hazardov

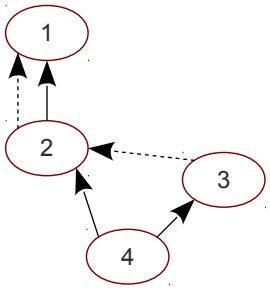
Ďalšou dôležitou položkou je objekt *HazardRec* triedy *ScheduleHazardRecognizer*, ktorá je deklarovaná v súbore *ScheduleHazardRecognizer.h*, konkrétna implementácia sa použije zo *ScoreboardHazardRecognizer.cpp/h*. *HazardRec* slúži pre odhalenie štrukturálnych hazardov. Obsahuje v sebe binárne pole, kde každý bit je priradený niektorej funkčnej jednotke. Toto pole je uložene v ďalšom poli, ktoré predstavuje obsadenosť funkčných jednotiek v ďalších cykloch. Môžme si to predstaviť tak, že na nultej pozícií je binárne pole obsadenosti jednotiek cyklu, pre ktorý zrovna plánujeme. Na prvej pozícií je binárne pole obsadenosti funkčných jednotiek v nasledujúcom cykle ...

Keď chceme vybrať inštrukciu do *bundle* pomocou metódy prebehne dotaz na *HazardRec* či by bolo možné vložiť inštrukciu do binárnych máp, teda či funkčné jednotky v potrebných cykloch sú voľné. Týmto spôsobom zabezpečíme správne vytvorenie *bundle*, čo sa týka štrukturálnych hazardov a tak isto zamedzíme hazardom medzi *bundle*. Vždy hľadáme také inštrukcie, ktoré je možné do binárnych máp vložiť. Keď naplníme maximálny počet inštrukcií v *bundle* alebo vyberieme všetky dostupné inštrukcie, do binárnych máp vložíme požiadavky na zdroje a pomyslene tikneme hodinami pomocou metódy *AdvanceCycle()*, čím sa v poli presunie binárna mapa z pozície 1 na pozíciu 0 ...

Stavba grafu závislostí

Na to, aby nevznikli medzi inštrukciami dátové hazardy sa stará spomínaná fronta *AvailableQueue* na základe vytvoreného grafu závislostí medzi inštrukciami. Implementácia stavby grafu je v súbore *ScheduleDAGInstr.cpp*, konkrétne v metóde *buildSchedGraph()*. V prvej časti algoritmu sa spracovávajú dátové závislosti. Čiže ak niektorá inštrukcia načítava hodnotu z registru číslo 4 a iná inštrukcia do tohoto registru zapisuje musí byť medzi nimi orientovaná hrana v smere od inštrukcie, ktorá sa vyskytla v pôvodnom programe neskôr ku inštrukcií, ktorá sa vyskytla skôr. Ide o to, aby pre tieto dve inštrukcie bolo zachované programové poradie.

Druhá časť algoritmu sa venujeme závislostiam medzi *store* a *load* inštrukciami. Používa sa alias analýza a ďalšie techniky, aby sa hrany medzi uzly nedávali zbytočne, ale len pre inštrukcie pre ktoré je to nutné. Na obrázku 4.7 je malá ukážka grafu závislostí medzi inštrukciami 1 až 4 z príkladu z obrázku 4.6.



Obrázok 4.7: Príklad grafu závislostí medzi inštrukciami.

Medzi inštrukciami číslo 4 a 3 je dátová závislosť, lebo inštrukcia *call* aj keď to nie je v ukážke napísané využíva všetky registre procesoru (je to z dôvodu toho, aby sa inštrukcia *call* nevolala v čase rozpracovaných a neuložených výsledkov). Z podobných dôvodov sa vytvorí závislosť medzi inštrukciami 4 a 2. Medzi inštrukciami 3 a 2 je nakreslená prerušovaná šípka. Nejde tu priamo o dátovú závislosť, lebo inštrukcia 3 nepotrebuje výsledok inštrukcie 2, ale nie je chcené, aby inštrukcia 3 vo výsledku predbehla inštrukciu 2. Kvôli tomu, že inštrukcia zapisuje do registru *regs_4* a inštrukcia 2 z neho číta. Ak by si inštrukcie vymenili poradie tak 2 by načítala zlý výsledok. Inštrukcia 2 je spojená s inštrukciou 1 obidvoma typmi hrán.

V príklade sú rozlíšené len dva typy hrán, ale obecne hrana môže mať niekoľko typov. Deklaráciu a teda aj rôzne typy je možné nájsť v *ScheduleDAG.h*. Okrem toho, že hrany môžu byť rôznych typov, môžu niesť informáciu o dĺžke prevádzania inštrukcie, na ktorú ukazujú. Tento súbor som modifikoval kvôli stavbe grafu závislostí nad superblokmi. Je

pridaná možnosť, že hrana typu *order* môže mať nulové oneskorenie, čo sa hodí pre modelovanie závislostí keď staviame graf závislostí nad väčšími celkami kódu na čo LLVM nie je moc prispôsobené. Táto činnosť je popísaná v sekcií 4.6.

Plánovanie

Samotné plánovanie potom prebieha tak, že na začiatku vložíme do *AvailableQueue* uzly, ktoré majú splnené svoje závislosti, teda v príklade z obrázku 4.7, by sa bola vložila inštrukcia 1. V ďalšom kroku prechádzame touto frontou a dotazujeme sa *HazardRec* či by naplánovaním vybraného uzlu došlo k hazardu. Buď vyberieme dostatočný počet inštrukcií pre úplne zaplnenie *bundle* alebo sa nám minú dostupné inštrukcie. Vybraté uzly zatiaľ vkladáme do vektoru, ktorý abstrahuje nad *bundle*. Je dôležite poznamenať, že v tomto momente nám nezáleží na správnom umiestnení inštrukcií v *bundle*. Je dôležité, aby v bundle boli len tie inštrukcie, ktoré nemajú medzi sebou žiadne závislosti a neprichádza pri ich vykonávaní ku štrukturálnym hazardom a vyhovujú obmedzeniam nad *bundle*.

V poslednej modifikácií implementácií sme jeden druh hazardu v *bundle* povolili. Jedná sa o *RAW* (Read After Write) konflikt. Predstavme si situáciu z obrázku 4.7 a teda 4.6. Sústreďme svoju pozornosť na inštrukcie 2 a 3. V predchádzajúcom texte som tvrdil, že inštrukcia 3 nemôže predbehnúť inštrukciu 2, čo je pravda, ale môžu byť naplánované v jednom *bundle*. Medzi inštrukciami dochádza k RAW konflitku na základe *regs_4*. Ak sú inštrukcie načítané v jednom takte tento konflikt nevadí, lebo v prvej fáze spracovania inštrukcií po dekódovaní sa načítajú hodnoty operandov, ktoré sú zatiaľ nemodifikované a teda správne. Potom po dokončení inštrukcií sa zapíšu výsledky naspäť do registrov.

Za zmienku stojí aj práca s *SchedulingBoundary* inštrukciou. Často je možné, že napríklad inštrukcia absolútneho skoku nevyužíva žiaden register a v grafe závislostí zostane nespojená zo žiadnou inou inštrukciou. V tomto prípade sa môže stať, že inštrukcia sa v *AvailableQueue* objaví hneď v prvých iteráciách plánovania a naplánovať skokovú inštrukciu skôr, ako by boli naplánované ostatné inštrukcie v regióne by viedlo k hrubej chybe. Tento problém riešim tak, že pri výbere regiónu pre plánovanie detekujem či región obsahuje *SchedulingBoundary* inštrukciu a ak áno uložím si odkaz na ňu. Potom neskôr, keď sa objaví v *AvailableQueue* odložím si ju bokom. Po vybratí inštrukcií algoritmus obsahuje celkom zložitú podmienku pre možnosť vložiť odloženú inštrukciu do *bundle*. Podmienka vloženia spočíva v tom, že musia byť spracované všetky ostatné inštrukcie z regiónu, vloženie odloženej inštrukcie by nespôsobilo hazard a je vôbec možné inštrukciu do *bundle* vložiť. Tu prichádza na rad obmedzenie v zmysle možných voľných pozícií inštrukcií v *bundle*. Skokové inštrukcie sa väčšinou v *bundle* môže nachádzať len na jednom mieste a iba jedna inštrukcia. Nemá asi veľký zmysel v jednom takte skákať v programe na dve rôzne miesta. Ak sú teda všetky podmienky splnené je možné odloženú inštrukciu vložiť do *bundle*, ak nie vložíme ju naspäť do *AvailableQueue* a algoritmus sa ju pokúsi spracovať v ďalšej iterácií plánovania.

Keď máme naplnený vektor uzlov, ktorý predstavuje *bundle*, je potrebné inštrukcie v tomto vektore zoradiť do správneho poradia, podľa obmedzenia architektúry. Backend v súbore *ArchGenInstrInfo.cpp* implementuje metódu *getAllPosInBundle()*, ktorá obsahuje zoznam všetkých inštrukcií. Metóda cez argumenty prijme opkód inštrukcie a odkaz na vektor, ktorý naplní možnými pozíciami pre dotazovanú inštrukciu. Naspäť v plánovaní algoritmus implementovaný v metóde *sortInstrInBundle()*, inštrukcie v *bundle* podľa týchto informácií zoradí. Najprv umiestňuje inštrukcie, ktoré majú obmedzený možný počet miest. Čiže inštrukciu, ktorá môže byť napríklad len na jednom mieste umiestni ako prvú. Je to

z toho dôvodu, aby inštrukcie, ktoré môžu byť na viacerých pozíciách v *bundle* zbytočne neobsadili jedinečné miesto inštrukcií, ktoré môžu byť napríklad len na jednej pozícií.

Po zoradení sa pristúpi ku samotnému naplánovaniu inštrukcií, čo znamená uzol vložiť do *PendingQueue*, na neobsadené miesto v *bundle* sa vloží *NULL* čo symbolizuje inštrukciu *NO–OP*. *PendingQueue* je implementovaná ako vektor odkazov na uzly *SUnit*. Ďalej je nutné prejsť cez naplánované uzly a vložiť ich potomkov do *AvailableQueue*.

Ak v danej iterácií plánovanie nebolo možné vybrať žiaden uzol do *bundle* naplánuje sa *bundle* plný inštrukcií *NO–OP*. Vyvolá sa pomyslený tik procesoru pomocou *AdvanceCycle* a inkrementuje premenná *CurCycle*, ktorá symbolizuje aktuálnu hĺbku v grafe závislostí inštrukcií. Implementáciu je možné nájsť v metóde *scheduleAlg()*.

Vytvorenie bundle

Výsledkom plánovania sú zoradené inštrukcie vo vektore. Tieto inštrukcie sú už predpripravené pre vytvorenie *bundle*. To znamená, že ak *bundle* má možnú veľkosť 4, tak každé štyri inštrukcie vo vektore reprezentujú jeden *bundle*.

Algoritmus pre vytváranie *bundle* prechádza zoradené inštrukcie a podľa veľkosti *bundle* ich vkladá z vektoru naspať do základného bloku. Je dôležité si uvedomiť, že je nutné ošetriť situácie, kedy sa narazí na *pseudo–inštrukciu*, napríklad sa tu môže objaviť *inline–assembler*. Takéto inštrukcie treba ošetriť tak, že ak sme mali rozpracovaný *bundle*, tak sa musí ukončiť, samostatne do základného bloku vložiť pseudo–inštrukcia a môže sa pokračovať ďalej ďalšou inštrukciou. Pseudo–inštrukcie musia byť umiestnené vždy mimo *bundle*, lebo napríklad v prípade inline–asembleru, pseudo–inštrukcia obsahuje už text, ktorý reprezentuje správne vytvorený *bundle*.

Po vložení všetkých naplánovaných inštrukcií späť do základného bloku a nad napríklad štvoricami inštrukcií je už vytvorená LLVM abstrakcia nad *bundle*, je potrebné do kódu vrátiť *debug–values*. Debug–values vznikajú pri preklade frontendom s nepovinným parametrom `-g` a obsahujú napríklad názov pôvodného súboru a číslo riadku, z ktorého sa daná inštrukcia preložila. Tieto informácie sú pri plánovaní odobraté od inštrukcií v čase stavania grafu. Sú uložené bokom, kde sa uchováva vždy debug–value a k nej odkaz na inštrukciu, na ktorú je naviazaná. V momente, keď vložíme naplánované inštrukcie späť do základného bloku, môžme vložiť späť aj debug–values. Debug–values je ale nutné vložiť vždy mimo *bundle*.

4.5 VliwNoopAdder

Jedná sa o jeden z dvoch priechodov, ktoré sa nachádzajú za priechodom, ktorý vytvára *bundle* v backende. Priechod pracuje už nad inštrukciami v *bundle*. Jeho úlohou je vyriešiť datové alebo štrukturálne hazardy medzi základnými blokmi. Pomocou plánovania popisovaného v predchádzajúcom texte, síce získame správne naplánovaný kód a riešenie hazardov, ale len v rámci základného bloku. Je však možná situácia, že do posledného *bundle* v základnom bloku sa naplánuje inštrukcia *load*, ktorej latencia je napríklad tri takty a hneď nasledujúci *bundle* v ďalšom základnom bloku obsahuje inštrukciu, ktorá výsledok *load* inštrukcie využíva. Dochádza tu k hazardu, ktorý je nutné vyriešiť. Súčasná implementácia v takýchto situáciách vkladá *bundle* plné inštrukcií NO–OP. Toto riešenie nie je ideálne a je možné, že preplánovaním základného bloku, by bolo možné obmedziť počet prázdnych *bundle*. Preplánovaním by však mohli vzniknúť ďalšie hazardy medzi základnými blokmi.

Riešením by bolo napísať priechod, ktorý bude korektne vytvárať *bundle*, vždy pre dvojice základných blokov.

Súčasná implementácia prechádza vždy dvojice základných blokov s tým, že rieši vzniknuté hazardy medzi nimi. Algoritmus prvým blokom prechádza a vkladá jednotlivé inštrukcie do detektoru hazardov, ktorý je použitý rovnaký ako v priechode *PreEmitVliwPass*. Tiež je implementovaná jednoduchá mapa, ktorá detekuje dátové hazardy. Ak teda ku hazardu dôjde, vkladá sa potrebné množstvo prázdnych *bundle*, kým sa hazard neodstráni.

Dvojice základných blokov sa nájdu tak, že sa prechádza základný blok a ak sa nájde inštrukcia skoku, ako druhý základný blok spracujeme cieľový blok inštrukcie. Ak inštrukcia skoku bola s podmienkou je na konci základného bloku bude buď inštrukcia absolútneho skoku alebo druhý základný blok sa zoberie blok, ktorý nasleduje ďalej v programovom poradí.

4.6 Vytvorenie a umiestnenie globálneho plánovania

Globálne plánovanie, ktoré som vytvoril je písane ako priechod do backendu. Očakáva, že vstup bude medzikód, ktorý má vytvorené superbloky. Výsledkom sú naplánované inštrukcie, ktoré môžu zmeniť svoje umiestnenie v programe a to nie len v rámci jedného základného bloku, ale svoju pozíciu môžu zmeniť v rámci superbloku. S tým, že presúvam inštrukcie cez skokové inštrukcie prináša zo sebou nutnosť generovať kompenzačný kód.

LLVM umožňuje do prekladu backendu vstúpiť programátorovi a vkladať vlastné priechody na niekoľkých miestach. Jedno z miest je pred alokáciou registrov a po prevedení medzikódu na inštrukcie danej architektúry. Toto je presne miesto, kde by plánovanie tohoto typu malo byť umiestnené. Problém pri testovaní nastal s tým, že pre niektoré súbory jedna z analýz potrebná pre funkciu plánovania (*LiveIntervalAnalysis*) zmenila kód takým spôsobom, že výsledný kód bol po preklade chybný. Z tohto dôvodu som vytvoril vlastný vstup do štruktúry prekladu backendu na miesto, kde prevádzať všetky analýzy potrebné pre plánovanie bolo už bezpečné. Zásah do LLVM kódu je možné nájsť v *lib/CodeGen/Passes.cpp*. Výsledkom je, že programátor môže použiť vo svojom backende metódu *addSuperBlockPreRegAlloc()* a vložiť do nej svoje priechody. Priechody sa vložia pred alokáciu registrov, ale už na miesto, kde použitie *LiveIntervalAnalysis* je bezpečné.

Medzi súbory backendu boli prenesené súbory implementované abstrakciu nad superblokmi *Superblock.h, BunchOfSuperBlocks.cpp, BunchOfSuperBlocks.h*. Tieto súbory síce funkcionalitou sú zhodné zo súbormi z *opt-superblock*, ale implementačne sú tu rozdiely, lebo backend pracuje už nad objektmi s prefixom *Machine*. Po načítaní základné bloku do superbloku je možné, že superblok bude poškodený. Nemôžme zaručiť, že užívateľ nepoužil optimalizáciu ešte nad medzikódom, ktorá superbloky rozbije alebo aj samotný backend používa už spomínaný priechod *licm (Loop invariant code motion)*, ktorý tým, že predradí pred cyklus ďalší základný blok, tak môže poškodiť superblok. Algoritmus po načítaní prejde superbloky ak nájde nejaký základný blok, ktorý porušuje definíciu superbloku, superblok na tomto bloku rozdelí na dve časti. Tiež sa môže stať, že niektorý zo základných blokov bude prázdny. Na tento prípad ďalšie algoritmy nie sú pripravené a musí sa to ošetriť pri načítaní superblokov. Implementácia týchto situácií je v metóde *fixSuperBlocks()* v súbore *lib/CodeGen/BunchOfSuperBlocks.cpp*.

Samotný priechod pre (nie len) globálne plánovanie je implementovaný v súbore *SuperBlockScheduling.cpp*. V súbore sú implementované dve triedy jedna pre samotný priechod a druhá pre implementáciu plánovania. Podobne ako priechod, ktorý vytvára *bundle*. Priechod nemusí plánovať len superbloky, je možné do neho vložiť celý základný blok. Táto

funkcionalita je prospešná hlavne pre základné bloky, kde je možné previesť tzv. virtuálny bundling. Tento pojem sme zaviedli pri vývoji a označuje činnosť, ktorá je bližšie popísaná v sekcií 4.9.

4.6.1 Tvorba grafu nad superblokom

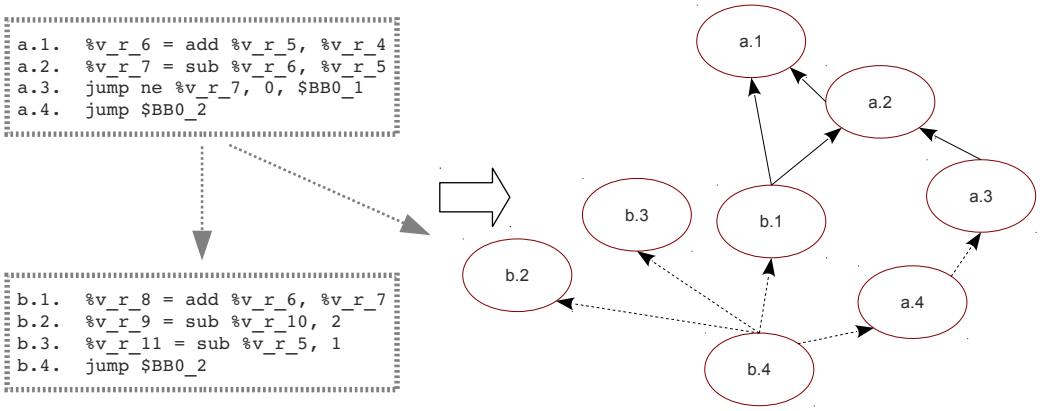
Po načítaní základných blokov do superbloku sa nad superblokom (superblok môže tiež obsahovať len základný blok) s inštrukcií vytvorí acyklický graf závislostí. LLVM nemá algoritmus, ktorý by korektne postavil graf z väčších častí ako sú regióny s jednou *schedule boundary* inštrukciou. Preto som vzal algoritmus popisovaný v sekcií 4.4.2 a prepracoval ho tak, aby bolo možné vytvoriť graf nad ľubovoľným počtom základných blokov.

Keď staviame acyklický graf závislostí nad viacerými základnými blokmi musíme v prvom rade zachovať poradie základných blokov. Inštrukcie skoku, volania (resp. všetky *schedule boundary* inštrukcie) musia zostať vo svojich materských základných blokoch. Zároveň neskôr pri plánovaní, kde sa už pracuje s grafom, neskôr zoznamom uzlov, musíme byť schopný nejakým spôsobom identifikovať základne bloky. Preto som sa rozhodol pre opatrenie, že vždy posledná inštrukcia v rámci základného bloku (väčšinou je to inštrukcia skoku, ale nemusí byť) bude pevná, teda nebude ju možné zo základného bloku odstrániť a bude identifikovať koniec základného bloku. Takýmto spôsobom po plánovaní, keď mám inštrukcie zoradené v zozname a nemám nejaké implicitné informácie, kde začína a kde končí ktorý základný blok, dokážem identifikovať týmto omedzením koniec každého základného bloku. Pre tvorbu grafu to znamená, že posledné inštrukcie základných blokov budú medzi sebou spojené, čím sa vytvorí závislosť a modeluje to omedzenie, že posledná inštrukcia z potomka sa nikdy nepresunie pred poslednú inštrukciu z predka. Toto rozhodnutie sebou prináša aj určitú nevýhodu a to takú, že nie každý základný blok musí nutne v tomto čase prekladu končiť skokovou inštrukciou. Tým, že ju prehlásim za poslednú a nepremiestniteľnú inštrukciu zamedzím tomu, aby sa vôbec nejak efektívne plánovala.

Ďalšiu závislosť musíme vytvoriť medzi všetkými *scheduling boundary* inštrukciami. Aby nebolo možné, že niektorá inštrukcia *call* sa presunie do iného základného bloku a podobne. To by mohlo viesť k veľkej chybe. Tak isto nie je možné dopustiť, aby niektorá inštrukcia nemala žiadne závislosti. Ak by takáto inštrukcia vznikla vytvorím závislosť medzi ňou a poslednou inštrukciou jej domovského základného bloku. Tým vznikne obmedzenie, že daná inštrukcia sa nesmie presunú do nasledujúceho základného bloku. Tejto tématike sa ešte budem venovať v sekcií o generovaní kompenzačného kódu (4.6.3), zatiaľ len naznačím, že ide o chcené obmedzenie.

Špeciálnu pozornosť si zaslúžia aj inštrukcie *call*. V tomto momente prekladu programu okolo inštrukcií *call* sú pseudo–inštrukcie *CALLSTART* a *CALLEND*, medzi ktorými môžu byť inštrukcie, ktoré by nemali opustiť túto ohraničenú oblasť. Preto je potreba umelo vytvoriť závislosti medzi inštrukciami a zároveň ak niektorá inštrukcia medzi *CALLSTART* a *CALLEND* má závislosť na niektorú inštrukciu mimo, je potrebné túto závislosť skopírovať na inštrukciu *CALLSTART*.

Majme príklad na obrázku 4.8. V ľavej časti obrázku sú zobrazené dva základné bloky, ktoré tvoria superblok. Každý riadok je označený, označenie priraďuje pozíciu vo vytvorenom grafe. Prefix `v_` znamená, že sa jedná o virtuálne registre. V tomto momente prekladu je prekladaný program ešte stále v SSA forme a obsahuje *phi* inštrukcie. Čo nám do značnej miery uľahčuje prácu pri generovaní kompenzačného kódu. Pre jednotlivé inštrukcie cieľ prevádzanej operácie je pred znakom priradenia, potom nasleduje názov operácie a dva operandy. V pravej časti obrázku je vytvorený graf. Algoritmus vytvára graf od poslednej



Obrázok 4.8: Príklad grafu závislostí medzi inštrukciami nad superblokom.

inštrukcie v superbloku. V prvej iterácií zoberieme inštrukciu skoku `b.4` a vložíme ju do grafu. Potom sa postupne do grafu vložia inštrukcie `b.3, b.2, b.1`. Nakoľko tieto inštrukcie nemajú zatiaľ v grafe žiadne závislosti automaticky sa vytvoria závislosti s poslednou inštrukciou ich domovského základného bloku. Ďalej sa spracuje inštrukcia `a.4`, ktorá je poslednou inštrukciou základného bloku a spojí sa inštrukciou `b.4`, tým sa zabezpečí, že inštrukcia `b.4` nepredbehne pri plánovaní inštrukciu `a.4` čo by viedlo k chybe. Ako ďalšia sa spracuje inštrukcia `a.3`, ktorá nakoľko je *boundary schedule* inštrukcia sa spojí s poslednou spracovanou *boundary schedule* inštrukciou, teda `a.4`. Potom sa spracujú inštrukcie `a.2` a `a.1` s tým, že sa vytvoria patričné dátové závislosti (plné šípky).

Okrem spomínaných vytváraných závislostí je nutné ošetriť ešte ďalšie prípady, kedy je lepšie danú situáciu vyriešiť konzervatívne a zamedziť možnosti presunúť inštrukciu mimo svoj základný blok. Zamedzenie presunu mimo základný blok sa vytvorí tak, že sa vytvorí závislosť medzi spracovávanou inštrukciou a nasledujúcou *boundary schedule* inštrukciou v protismere vykonávania programu. Medzi inštrukcie, ktoré je nutné ponechať v ich základných blokoch sú inštrukcie pri ktorých je možné, že dôjde k výnimke, napríklad inštrukcie *load* a *store*. Ďalej inštrukcie *phi* musia zostať vo svojom bloku. Pri testovaní som prišiel na to, že tiež inštrukcie, ktoré už obsahujú fyzický register je nutné nechať v ich domovskom bloku.

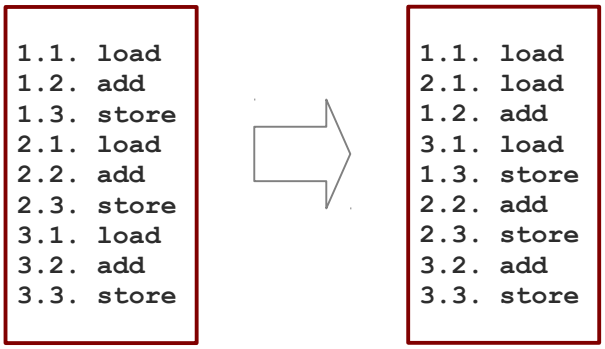
Algoritmus pre stavbu grafu nad superblokmi je uložený v súbore *ScheduleDAGSuperBlock.cpp*

4.6.2 Plánovanie

Algoritmus pre plánovanie spolieha na to, že graf závislostí bude obsahovať všetky potrebné závislosti a vo svojej réžií nerieši žiadne špeciálne situácie. Nad grafom závislostí sa prevádza *List Scheduling* s heuristikou *Critical path*. Obidve techniky sú popísané v sekcií 4.4.2.

Virtuálny bundling

Táto technika má do určitej miery nahradiť chýbajúce zreťazenie cyklov (popísané v sekcií 3.5.3). Pre vysvetlenie uveďme príklad na obrázku 4.9. Majme rozbalený cyklus v jednom základnom bloku v ľavej časti obrázku. Ak by sme aplikovali na takýto kód alokátor registrov



Obrázok 4.9: Príklad metódy virtuálny bundling.

potreboval by na celý kód len jeden register do ktorého by v prvej inštrukcií načítal, druhej pripočítal konštantu a v tretej hodnotu registru uložil do pamäte. Ďalšie tri inštrukcie by spravili to isté, síce s inou hodnotou, ale postačí iba jeden register. Pre architektúru VLIW by bol takýto prístup málo efektívny. Preto je lepšie počas plánovania použiť všetky obmedzenia ako keby (virtuálne) sa vytváral *bundle*. *Bundle* sa nemôžu vytvoriť kvôli tomu, lebo alokátor registrov môže pridať do kódu inštrukcie. Určite sa vloží kód pre prológ a epilóg funkcií. Takže fyzicky vytvárať *bundle* by bolo neefektívne. Ale použiť pri plánovaní ďalšie obmedzenia, tak aby sa inštrukcie zoradili ako keby do bundle môže podstatne zrýchliť prekladaný program.

Výsledok je možné vidieť v pravej časti obrázku. Tento kód si už z jedným registrom nevystačí a zvyšuje sa úroveň paralelizmu na úrovni inštrukcií.

Pri testovaní rôznych programov som narazil na problém v tom zmysle, že často LLVM a jeho *AliasAnalyse* nevie vždy odhaliť hlavne pri inštrukciách *load* a *store*, že dve inštrukcie nie sú na sebe závislé a určite neodkazujú na rovnaké miesto v pamäti. Nakoľko analýza túto skutočnosť nevie väčšinou správne detekovať vytvorí závislosti medzi inštrukciami a nie je ich možné presúvať, tak ako je to uvedené v príklade na obrázku 4.9. Dopomôcť musí sám užívateľ, ktorý môže pozmeniť prekladaný program tak, že na vhodné miesta vloží kľúčové slovo jazyka C `restrict`. Je nutné ho vložiť pri deklarácií funkcie a jej parametrov medzi hviezdičku a názov premennej napríklad obrázok 4.10. Programátor použitím

```
void func(int * restrict a);
```

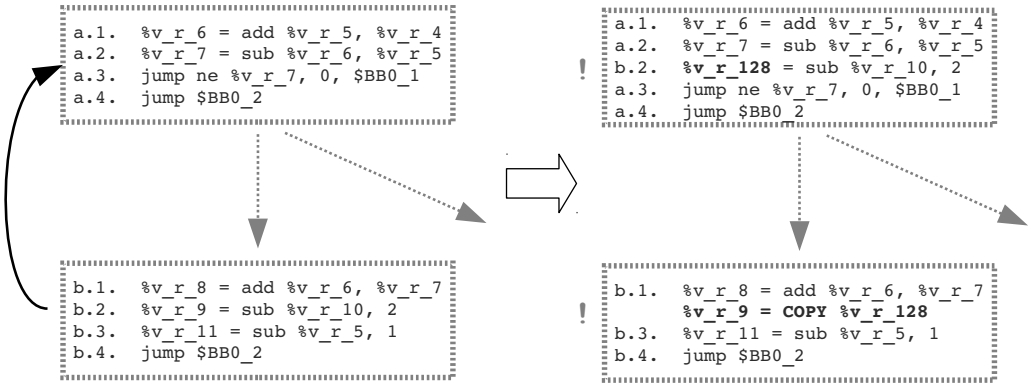
Obrázok 4.10: Použitie kľúčového slova restrict.

`restrict` zaručuje, že v rámci danej funkcie nebude označená premenná ukazovať na rovnakú pamäť ako akákoľvek iná premenná vo funkcií. Podľa niektorých ukážok je možné `restrict` použiť aj na premenné deklarované v rámci funkcie, ale LLVM to nepodporuje. Jediné možné použitie na ktoré sme prišli je, že LLVM podporuje deklaráciu pointru ako parameter funkcie.

4.6.3 Generovanie kompenzačného kódu

Generovanie kompenzačného kódu uľahčuje niekoľko faktorov. Napríklad, že kód ešte nemá alokované fyzické registre a v podstate ešte stále je v SSA forme. Obmedzenia pri tvorbe grafu a použitý algoritmus pre plánovanie umožní presúvanie inštrukcií iba jedným smerom. Čiže ak inštrukcia vyhovuje všetkým kritériám je ju možné presunúť do základného bloku, ktorý v programovom poradí skôr. Opačným smerom to kvôli tomu ako je graf postavený nepôjde. Týmito krokmi sme inak celkom zložitú časť generovania kompenzačného kódu dosť zjednodušili.

Uvediem príklad na obrázku 4.11. Majme superblok obsahujúci dva základné bloky a



Obrázok 4.11: Ukážka vloženia kompenzačného kódu.

plánovanie presunulo inštrukciu `b.2` z druhého bloku do prvého. Uskutočnil sa tu presun cez bočný výstup čo si vyžaduje test, či výsledok inštrukcie je použitý mimo superblok. Ak áno kompenzačný kód je nutný. Ak nie, tak poradie inštrukcií v rámci superbloku je po plánovaní správne, teda žiaden kompenzačný kód nie je potrebný. Kompenzačný kód sa vytvorí pomocou pseudo–inštrukcie *COPY* tak, že pôvodnej inštrukcii sa zmení cieľový virtuálny register a vytvorí sa nový. Na pôvodné miesto inštrukcie sa vloží pseudo–inštrukcia *COPY*, ktorá výsledok presunutej inštrukcie vloží do pôvodného registru v pôvodnom základnom bloku. Takýmto spôsobom sa ošetril prípad, kedy by sa použil bočný výstup, výsledok presunutej inštrukcie by sa nikdy nepoužil, čiže výsledný program je korektný.

Implementačne je situácia trochu zložitejšia. Výsledok plánovania je vektor (zoznam) zoradených inštrukcií. Ako som už vyššie spomínal pri generovaní kompenzačného kódu musíme v tomto vektore určitým spôsobom rozlíšiť, kde ktorý základný blok končí. Tomu nám napomáha znalosť posledných inštrukcií v pôvodnom nenaplánovanom základom bloku. Lebo omedzenia pri tvorbe grafu zaručujú, že táto inštrukcia bude posledná aj v novo–naplánovanom základnom bloku. Takže, pohybujeme sa cez vektor inštrukcií a testujeme či sa inštrukcia nachádza vo svojom základnom bloku alebo nie. Ak nie, testuje sa či výsledok inštrukcie je použitý mimo superblok, ak áno generuje sa kompenzačný kód vyššie popísaným spôsobom. Implementácia je v metóde *genCompensatoryCode()* v súbore *SuperBlockScheduling.cpp*.

4.6.4 Emitovanie naplánovaných inštrukcií

Na začiatku je nutné všetky inštrukcie z pôvodných základný blokov vymazať. Keď sú základné bloky prázdne je možné do nich postupne vkladať naplánované inštrukcie. *Phi*

inštrukcie sú inštrukcie, ktorým sa musí venovať špeciálna pozornosť. Keď sa pri vkladaní inštrukcií do základných blokov superbloku narazí na *phi* inštrukciu je nutné ju vložiť vždy na začiatok domovského bloku.

Ďalšiu pozornosť si zaslúži vrátenie *debug–values*, ktoré boli odobraté z kódu počas stavby grafu. Opäť treba dať pozor na *phi* inštrukcie, aby *debug–value* sa nevložila medzi ne, inak by ďalšie priechody s tým mali problém.

4.6.5 Vymazanie prázdnych základných blokov

Poslednou metódou volanou nad naplánovanými superblokmi je `tryEraseEmptyBB()`. Implementácia vyhľadáva základný blok v superbloku, ktorý obsahuje len jednu inštrukciu a to je inštrukcia absolútneho skoku. Takýto základný blok po plánovaní môže ľahko vzniknúť. Ak sa vymazanie bloku podarí je nutné upraviť adresy skoku a *phi* inštrukcie jeho nasledovníkov a predchodcu.

4.7 Používanie vytvorených nástrojov

V nasledujúcom texte sa venujem trom modelovým situáciám ako je možné postupovať pri preklade zdrojového súboru z jazyka C do jazyka symbolických adries danej architektúry. Presné použitie nástrojov je uvedené len pri nástrojoch, ktoré som vytváral, ide skôr o poradie použitých nástrojov.

Majme zdrojový kód v jazyku C a chceme ho preložiť do jazyku symbolických adries s využitím implementovaných nástrojov. Postup prekladu ide podľa nasledujúcich bodov:

1. preklad pomocou frontendu `clang` do LLVM–IR,
2. použitie `opt-superblock -i inputFile.ll -u levelOfUnroll -o readyForSB.ll`,
3. vloženie inštrumentácie
   `opt -q -f -insert-optimal-edge-profiling readyForSB.ll -o instrCode.bc`,
4. preklad backendom do asembleru, preklad asembleru do binárnej formy, aplikácia linkru, výsledok je binárny súbor určený pre *Codasip* simulátor,
5. po simulácií vznikne súbor `llvmprof.out`,
6. použitie `opt-superblock -i readyForSB.ll -p llvmprof.out -s -o SB.ll`,
7. preklad backendom do asembleru, preklad asembleru do binárnej formu, aplikácia linkru a výsledný binárny súbor.

Pri preklade a hľadaní spôsobu ako zrýchliť prekladaný program, môžme zistiť, že aplikácia `opt-superblock` z bodu 2, teda rozbalenie cyklov program nezrýchli, ale spomalí. V takom prípade môžme skúsiť zameniť tento krok za nasledujúci príkaz. Dôležité je aplikovať priechod `reg2mem` ako posledný priechod.

2. použitie `opt -inline -simplifycfg -reg2mem inputFile.ll -o readyForSB.ll`

Pri rozsiahlych programoch je dobré aplikovať chcené optimalizácie len nad niektorými funkciami. Zoznam funkcií získame tak, že program bez akýchkoľvek optimalizácií preložíme a odsimulujeme. Simulátor musí byť vygenerovaný s možnosťou získavať profil. Takže po prvej simulácií získame profil zo simulátoru a nájdeme v ňom funkcie, kde simulácia strávila

najviac času. Mená týchto funkcií uložíme do textového súboru ako bolo uvedené v príklade na obrázku 4.3. Potom použitie programu `opt-superblock` bude nasledovné.

2. použitie
```
opt-superblock -i inputFile.ll -f funcNameFile -u levelOfUnroll
               -o readyForSB.ll
```

6. použitie
```
opt-superblock -i readyForSB.ll -f funcNameFile -p llvmprof.out
               -s -o SB.ll
```

Ak optimalizujeme len niektoré funkcie je možné znovu použiť simulátor generovaný tak, aby získaval profil o prevedených funkciách. Je možné, že program trávi najviac výpočetného času v iných neoptimalizovaných funkciách. V takomto prípade môžme mená týchto funkcií pridať do súboru so zoznamom často prevádzaných funkcií a celý preklad spustiť znovu.

4.7.1 Súhrn známych obmedzení

Ako už bolo spomenuté použitie optimalizácií nemusí vždy viesť ku zrýchleniu programu. Napríklad rozbalenie už rozbaleného cyklu môže viesť k vložení ďalších základných blokov do prekladaného programu a celkovému spomaleniu.

Hlavným cieľom vyššie uvedených optimalizácií je zväčšiť základné bloky alebo vytvoriť väčšie regióny inštrukcií pre plánovanie. Teda, aby plánovanie si mohlo vyberať z čo najväčšieho počtu inštrukcií a tým lepšie využiť paralelizmus na úrovní inštrukcií. Ak teda optimalizáciami príliž zväčšíme kód v budúcnosti môže byť problém s inštrukčnou *cache* pamäťou. Model na úrovni cyklov procesoru *Codix-vliw* v čase vývoja ešte nebol implementovaný. V súčasnom modeli existuje ale obmedzenie, ktoré pri príliž veľkom kóde tiež spomalí výsledný program. Inštrukcia podmieneného skoku má obmedzenú veľkosť pre adresu miesta skoku. Ak veľkosť tejto adresy nepostačuje podmienka sa z neguje a pridá sa *bundle* obsahujúce iba jeden absolútny skok. Tento problém by sa dal vyriešiť napríklad zakódovaním *bundle*, ktorý obsahuje samé inštrukcie *NO-OP* ako jednu inštrukciu, čím sa nám zmenší preložený kód o tri inštrukcie.

Ďalším obmedzením je nedostatočná sila alias analýzy v LLVM. Respektíve nutnosť zasahovať do prekladaného programu a v vkladať `restrict` na vhodné miesta.

Niekoľko problémov v rámci implementácie stavby grafu, plánovania a vytvárania superblokov som riešil konzervatívne. Na prvom mieste musí byť funkčnosť výsledného programu na strane druhej ďalšou prácou by bolo možné niektoré obmedzenia vyriešiť inak.

4.8 Testovanie a zhodnotenie výsledkov

Vývoj nástrojov v týme *Lissom* obsahuje aj spúšťanie testov každú noc. Moje rozšírenia sú súčasťou balíku nástrojov, ktoré *Codasip toolchain* obsahuje, takže sa testy púšťajú aj pre VLIW architektúru. Testy sú hlavne zamerané na funkcionalitu. Implementujú rôzne základné konštrukcie jazyka C, ale aj zložitejšie komplexné benchmarky. Súčasťou testov sú aj benchmarky nad celými číslami z LLVM testsuity.

Z hľadiska rýchlosti som implementáciu testoval na niekoľkých algoritmoch, piatich algoritmoch z benchmarku *MiBench*[1], dekódovanie vidia s algoritmom *mpeg2* a dekódovanie videa z algoritmom *mpeg4*.

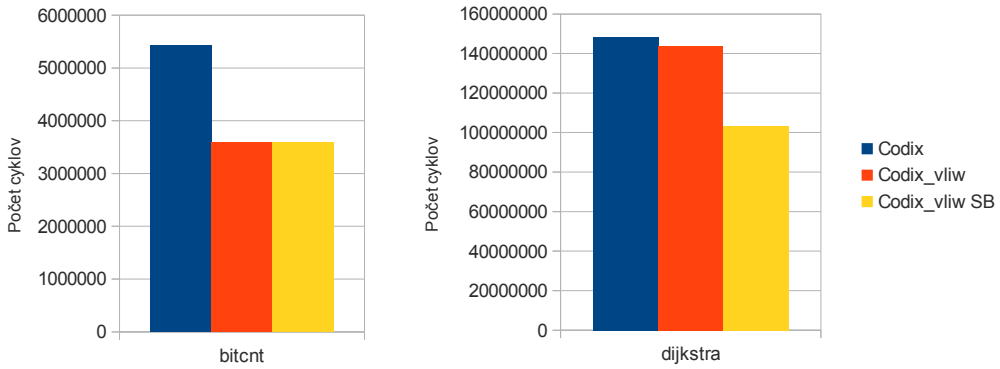
[1] Zdrojové súbory sú dostupné na http://www.eecs.umich.edu/mibench/.

Pri testovaní rýchlosti som algoritmus najprv preložil a spustil na architektúre *Codix*, ktorá je materskou architektúrou VLIW architektúry *Codix-vliw*. V architektúre *Codix* bola upravená latencia skokovej inštrukcie z hodnoty 3 na hodnotu 1, tak ako to je v *Codix-vliw*. Potom som program preložil na architektúre *Codix-vliw* bez vytvorenia superblokov a s profilovaním, vytvorením superblokov a globálnym plánovaním. Preklad pre *Codix-vliw* neobsahoval priechod *VliwNoopAdder*, kvôli tomu, že architektúra *Codix* obdobný priechod síce má, ale neošetruje všetky prípady tak striktne. Tiež v rámci bundlingu neboli povolené *RAW* hazardy kvôli tomu, že v čase testovania to simulátor ešte nepodporoval. Všeobecne modely a *Codasip* nástroje sa neustále vyvíjajú a nové testovanie s najnovšími zmenami v modeli by priniesli iné výsledky. Uvádzam teda výsledky, ktoré boli platné v čase vývoja s nástrojmi a modelmi, ktoré som mal k dispozícií.

Výsledky sú uvedené v tabuľke 4.12 a vizualizované na grafoch 4.13, 4.14, 4.15 a 4.16. Skratka *SB* znamená preklad s vytvorením superblokov a použitím globálneho plánovania.

	Codix	**Codix_vliw**	**Codix_vliw SB**
	počet cyklov		
bitcnt	5 426 884	3 584 064	3 583 864
dijkstra	148 400 681	143 951 865	103 334 464
quicksort	49 107 706	47 826 979	47 825 816
sha	3 727	1 790	1 759
strsearch	18 414 692	14 980 270	6 479 571
mpeg2	10 855 906	6 923 595	5 773 552
mpeg4	2 851 768 749	2 214 848 739	2 188 670 678

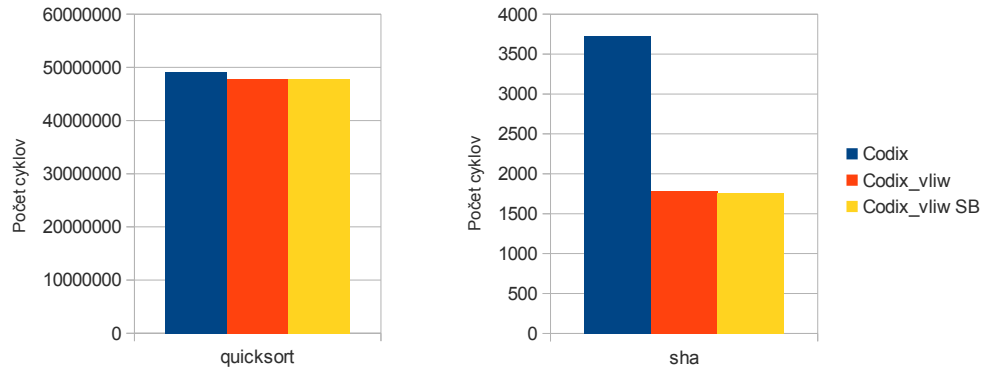
Obrázok 4.12: Tabuľka výsledkov testov zrýchlenia.



Obrázok 4.13: Výsledky zrýchlenia pre programy bitcount a dijkstra.

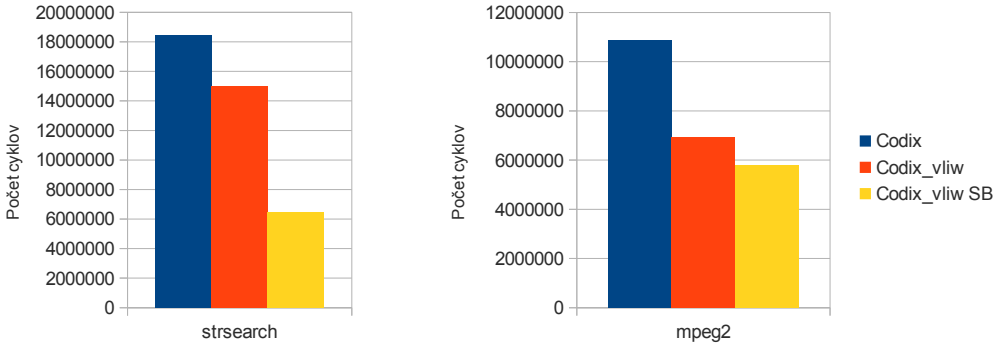
Program *bitcnt* bol prekladaný bez prvého použitia programu *opt–superblock*, lebo frontend už rozbalil cykly optimálne a ďalšie rozbalenie cyklov spomalilo výsledný program. Vytvorenie superblokov v tomto prípade viedlo len k nepatrnému zrýchleniu. Program *dijkstra* bol prekladaný z dodatočným rozbalením cyklov pomocou programu *opt–superblock*. Je zaujímavé, že použitie VLIW architektúry oproti RISC *Codixu* neprinieslo veľké zrýchlenie.

Vytvorenie superblokov na základe profilu a globálne plánovanie však zrýchlenie prinieslo.



Obrázok 4.14: Výsledky zrýchlenia pre programy quicksort a sha.

Na programe *quicksort* použitie VLIW neprinieslo nejaké zjavné zrýchlenie a nepomohli tomu ani optimalizácie na základe profilu. Naopak pri programe *sha* použitie VLIW architektúry prinieslo rapídne zrýchlenie. Vytvorenie superblokov a globálne plánovanie program ešte malinko zrýchlili.

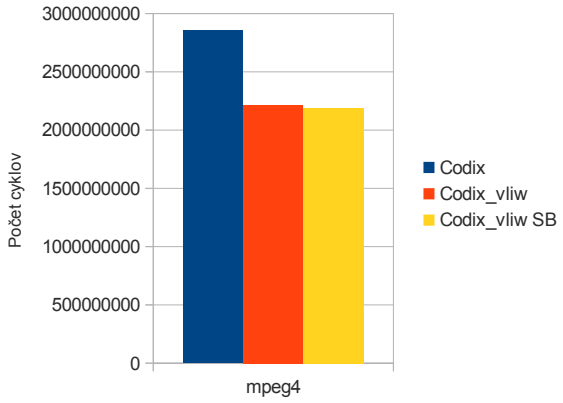


Obrázok 4.15: Výsledky zrýchlenia pre programy stringsearch a mpeg2.

Program *stringsearch* je ideálnou ukážkou zrýchlenia programu, keď sa použijú superbloky a globálne plánovanie. Do programu som zasahoval a to tak, že som pridával `restrict` na každé vhodné miesto. Tak isto program *mpeg2* sa zrýchlil nie len použitím VLIW architektúry, ale aj použitím mnou navrhnutých a implementovaných rozšírení.

Prekladu programu *mpeg4* som pri testovaní venoval najdlhší čas. Použitie VLIW architektúry prinieslo zrýchlenie, ale dodatočné rozbalenie cyklov nie. Takže prvé použitie programu *opt–superblock* v tomto prípade odpadá. Problém bol, že spomalenie prinieslo aj použitie superblokov. Program *mpeg4* je už komplexnejší program a tu sa preukázalo použitie optimalizácií len na vybraté funkcie ako cesta síce k nepatrnému, ale predsa zrýchleniu.

Pri preklade je možné použiť niekoľko postupov a niekoľko priechodov, ktoré ponúka samotné LLVM. Tiež je možné upravovať zdrojový program. Veľakrát treba experimentovať, lebo pre každú aplikáciu môže priniesť iný postup lepšie alebo horšie výsledky. Z testovania



Obrázok 4.16: Výsledky zrýchlenia pre programy mpeg4.

je možné vidieť, že výsledné zrýchlenie závisí od aplikácie a od toho ako je napísaná. Ak je možné vhodne doplniť `restrict` do zdrojového kódu, použitie VLIW architektúry a globálneho plánovania nad superblokmi môže priniesť pozoruhodné zrýchlenie. Pre veľké programy nie je vždy jednoduché zdrojový program upravovať. Rozhodne je, ale dobré aplikovať optimalizácie, ktoré zväčšujú zdrojový program len na vybrané funkcie.

4.8.1 Obsadenosť slotov v bundle

Ak chceme naplno využiť výhody *Codasip frameworku* a nevyvíjať len prekladané programy, ale aj meniť samotnú architektúru, je dobré mať informáciu o využití jednotlivých slotov v *bundle*. Nakoľko architektúra *Codix-vliw* neobsahuje kompresiu *bundle*, na voľné miesta sa vkladá inštrukcia *NO–OP*. Jednoduchou metrikou je možné zistiť využitie jednotlivých slotov veľmi dlhých inštrukcií pri vykonávaní programu simulátorom. Simulátor spočíta všetky neprázdne inštrukcie pre každý slot a porovnáme ich s počtom cyklov potrebných pre simuláciu celého programu. Výsledky pre testované programy sú zobrazené v tabuľke na obrázku 4.17.

	pomer využitých slotov v inštrukciách [%]			
	1 slot	**2 slot**	**3 slot**	**4 slot**
bitcnt	76	47	0	0
dijkstra	60	25	3	0
quicksort	62	1	0	0
sha	90	46	16	2
strsearch	72	40	0	0
mpeg2	69	20	5	3
mpeg4	59	14	4	3

Obrázok 4.17: Obsadenie slotov v bundle.

Z tabuľky je možné vidieť niekoľko informácií. Prvá, ktorá nemusí byť až tak zjavná je, že

programy obsahovali pomerne veľa úplne prázdnych *bundle*, potrebných prázdnych taktov, aby sa uvoľnili požadované jednotky (pre program *bitcnt* až 24% inštrukcií bolo prázdnych). Je to možné odvodiť z obsadenosti prvého slotu. Lebo každá inštrukcia v architektúre *Codix–vliw* môže byť v prvom slote. Najväčšou príčinou prázdnych *bundle* je inštrukcia *load*, ktorá ma latenciu 3. Riešením, môže byť použitie *cache* pamätí.

Ďalší záver, ktorý je možné z merania vyvodiť je, že pre niektoré aplikácie nemá zmysel 3 a 4 slot. Je samozrejme možné prehlásiť, že program ešte nie je optimálny a v optimálnom prípade by bolo možné sloty zaplniť viac. Nakoľko sa jedná o implementáciu niekoľkých úplných NP problémov pomocou heuristík a ku niektorým riešeniam problémov sa pristupovalo konzervatívne, ako autor súhlasím s týmto tvrdením. Na druhej strane vývojár pracuje s nástrojami, ktoré má dostupné a ak mu nástroje poskytnú dostatočný výkon, tak je možné zmeniť tvorenú architektúru, napríklad tým, že sa odoberie 3 a 4 slot. Tým sa zjednoduší model a samotný výsledný hardvér.

Kapitola 5

Záver

Práca zo začiatku rozoberá jazyk CodAl a Codasip framework, v ktorom je možné vyvíjať rôzne architektúry procesorov. Zameriava sa na objasnenie spôsobu generovania prekladača a jeho použitie.

Nakoľko spomínaný prekladač je založený na kompilačnej platforme LLVM, ďalšia časť práce pojednáva práve o nej. Opisuje medzikód *LLVM IR* a optimalizačné priechody, ktoré je možné nad ním previesť. Venujem sa hlavne tým častiam, ktoré som použil alebo zmenil pri vypracovaní tejto práce.

Ďalšia časť sa venuje samotnému návrhu konceptu zadania. Opisuje a ilustruje zmeny na úrovní medzikódu a v backende. Rozoberá štyri možné zoskupenia základných blokov pre umožnenie globálneho plánovania a vyberá z nich superbloky. Ďalej popisuje umiestnenie priechodu globálneho plánovania v backende, jeho pozície a algoritmus pre jeho realizáciu. Tiež sa venuje ďalším možným optimalizáciám cyklov a profilácií programu. Približuje aké možné profily LLVM podporuje a ako sa používajú.

Nasleduje samotná implementácia predchádzajúceho návrhu. Postupne je popísaná príprava pre tvorbu superblokov, ktorá sa venuje hlavne rozbaleniu cyklov a celkovým možnostiam ako zväčšiť základný blok. Popisuje zoradenie upravených LLVM optimalizačných priechodov a dôvod pre ich poradie. Prípravu pre tvorbu superblokov a samotné ich vytvorenie implementuje mnou vytvorený program *opt–superblock*. Program postupne aplikuje optimalizačné priechody hlavne priechod pre vytvorenie superblokov na základe profilu. Práca obsahuje detailné informácie o implementovanou algoritme a logike algoritmu.

Ďalej sa text venuje backendu a všetkými priechodmi, ktoré bolo nutné doplniť, aby bolo možné prekladať program pre architektúru VLIW pomocou *Codasip* nástrojov. Popisujem novú implementáciu priechodu pre vytváranie *bundle* a priechody, ktoré za ním nasledujú.

V práci popisujem umiestnenie a implementáciu globálneho plánovania nad superblokmi. Dôležitá časť je tvorba acyklického grafu závislostí inštrukcií nad základným blokom alebo superblokom, ktorú som doplnil a upravil tak, aby sa mohol použiť štandardný algoritmus *ListScheduling* bez nejakých väčších obmedzení. Globálne plánovanie musí obsahovať aj implementáciu kompenzačného kódu, ktorá je tiež popísaná.

Záverečná časť práce sa venuje použitiu implementovaných programov a rozšírení. Hlavne poradiu a logike použitých nástrojov. A tiež sa venuje testovaniu a zhodnoteniu výsledkov.

Prácu som koncipoval tak, aby popísala všetky potrebné základné techniky, pre tvorbu prekladača pre VLIW architektúry. V druhej časti popisujem svoju vlastnú implementáciu a rozšírenie LLVM. Ako svoj prínos považujem možnosť *Codasip frameworku* a hlavne *Codasip* kompilátoru prekladať program pre VLIW architektúry popísané v jazyku *CodAl*. Ďalším mojím prínosom je implementácia tvorby superblokov na základe profilu. Superbloky sa nemusia požiť len pre globálne plánovanie, ale tiež pre iné optimalizácie ako napríklad nachádzanie špeciálnych inštrukčných rozšírení.

Literatura

[1] LLVM Language Reference Manual [online]. [cit. 2012-12-31].
URL http://llvm.org/docs/LangRef.html

[2] clang: a C language family frontend for LLVM [online]. [cit. 2013-01-05].
URL http://http://clang.llvm.org/

[3] ApS Brno s.r.o: *Návrh procesoru Codix*.

[4] ApS Brno s.r.o: *Codal Manual*. 4.2 vydání, 2012.

[5] ApS Brno s.r.o: *Codasip ® Framework Manual*. 6.4 vydání, 2012.

[6] Ball, T.; Larus, J. R.: Efficient Path Profiling. In *In Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996, s. 46–57.

[7] Chekuri, C.; Johnson, R.; Motwani, R.; aj.: Profile-Driven Instruction Level Parallel Scheduling with Application to Super Blocks. In *IN PROCEEDINGS OF THE 29TH INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE*, 1996, s. 58–67.

[8] Deitrich, B. L.; mei W. Hwu, W.: Speculative hedge: Regulating compile-time speculation against profile variations. In *In Proceedings of the 29th International Symposium on Microarchitecture*, 1996, s. 70–79.

[9] Eichenberger, A. E.; Meleis, W. M.: Balance Scheduling: Weighting Branch Tradeoffs in Superblocks. In *PROC. 32 ND ANN. INT'L SYMP. MICROARCHITECTURE (MICRO32*, 1999, s. 272–283.

[10] Fisher, J. A.: Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, ročník 30, 1981: s. 478–490, ISSN 0018-9340.

[11] Gupta, R.; Mehofer, E.; Zhang, Y.: Profile Guided Compiler Optimizations. 2002.

[12] Lam, M.: Software pipelining: An effective scheduling technique for VLIW machines. 1988, s. 318–328.

[13] Lattner, C., Adve, V.: The LLVM Compiler Framework and Infrastructure Tutorial [online]. 2004-09-01 [cit. 2012-12-30].
URL http://llvm.org/pubs/2004-09-22-LCPCLLVMTutorial.html

[14] Mahlke, S. A., et al.: The Superblock: An effective technique for VLIW and superscalar compilation. *THE JOURNAL OF SUPERCOMPUTING*, ročník 7, 1993: s. 229–248.

[15] Mináč, T.: *Plánovač instrukcí překladače jazyka C pro VLIW architekturu.* Bakalářská práce, FIT VUT v Brně, 2011.

[16] Nicolau, A.: Percolation Scheduling: A Parallel Compilation Technique. Technická zpráva, Ithaca, NY, USA, 1985.

[17] Spencer, R., Henriksen, G.: LLVM's Analysis and Transform Passes [online]. [cit. 2012-12-31].
URL http://llvm.org/docs/Passes.html

Příloha A

Návod na použitie a obsah DVD

V tejto časti popisuje postup prekladu pridaných súborov a obsah pridaného DVD. Zdrojové kódy obsahujú pozmenené LLVM vo verzií 3.2. Kód je doplnený o backend *Codix–vliw*, implementáciu programu *opt-superblock* a tiež globálneho plánovania.

A.1 Preklad LLVM a použitie backendu

Zdrojové súbory sa najprv musia preložiť, ale skôr ako použijeme príkaz `make`, prevedieme konfiguráciu. Dopuručuje sa vytvoriť si nový priečinok, kde zadáme príkaz:

```
SOURCE_CODE/configure -prefix=‘pwd‘../install
```

`SOURCE_CODE` v našom prípade reprezentuje úplnú cestu do priečinku so zdrojovými súbormi prekladača LLVM. Použité sú len nutné voľby pre správnu konfiguráciu. Po skončení konfigurácie pokračujeme zadaním príkazov:

```
make
make install
```

Týmto spôsobom sa nám preloží a nainštaluje LLVM so všetkými nástrojami.

Aby sme preložili zdrojový súbor z vyššieho jazyka, napríklad jazyka C použijeme práve preložený nástroj z LLVM *clang*. Použitie je trochu zložitejšie, lebo vychádza z komplexných skriptov *Codasip frameworku*.

```
clang file.c -fno-builtin -emit-llvm -S -O3 -o file.ll -D__codix_ia__
            -ccc-custom-datalayout e-p:32:32:32-S64-a0:0:32-n32
          -D__mips_soft_float -nostdinc -DNO_UNALIGNED_LOADSTORE
                  -I toolchain/include -target codasip
```

Prepínač `-emit-llvm` značí, že výsledný kód bude určený pre backend LLVM. Závisí na ďalšom prepínači ak použijeme `-c` výsledný súbor s medzikódom bude *bitcode*. Do textovej podoby jazyka LLVM prevedieme zdrojový súbor použitím prepínača `-S`. Ďalšie prepínače sú uvedené len pre úplnosť, pre jednoduché programy je možné ich vynechať.

Keď je prekladaný program v medzikóde, je možné použiť nástroj *llc*. Tento nástroj slúži ako backend a jeho výstupom je program v jazyku symbolických adries danej architektúry, v našom prípade pre architektúru *Codasip–vliw*.

```
llc -march=codasip file.ll -o file.asm
```

Výsledok predchádzajúceho príkazu je zdrojový program v asembleri pre architektúru *Codix–vliw*. Program je možné ďalej spracovať ďalšíme nástrojami z *Codasip frameworku*.

A.2 Obsah DVD

Obsahom priloženého DVD sú:

- llvm-3.2-GlobSched/ – zdrojové súbory LLVM doplnené o backend *Codix–vliw*, *opt-superblock* a implementáciu globálneho plánovania,
- dp.pdf – elektrická verzia diplomovej práce,
- dp/ – obsahuje zdrojové súbory k písomnej správe.